Low-power epidemic communication in wireless ad hoc networks

Dobson, M.C.

2013

document version
Publisher's PDF, also known as Version of record

Low-power epidemic communication in wireless ad hoc networks

Matthew Carlson Dobson

VRIJE UNIVERSITEIT

Low-power epidemic communication in wireless ad hoc networks

ACADEMISCH PROEFSCHRIFT

ter verkrijging van de graad Doctor aan
de Vrije Universiteit Amsterdam,
op gezag van de rector magnificus
prof.dr. F.A. van der Duyn Schouten,
in het openbaar te verdedigen
ten overstaan van de promotiecommissie
van de Faculteit der Exacte Wetenschappen
op dinsdag 8 oktober 2013 om 11.45 uur
in de aula van de universiteit,
De Boelelaan 1105

door

Matthew Carlson Dobson

geboren te Massachusetts, Verenigde Staten

promotor: prof.dr.ir. M.R. van Steen
copromotor: dr. S. Voulgaris

members of the
thesis committee: prof.dr. Koen Langendoen, TU Delft
prof.dr. Konrad Iwanicki, University of Warsaw
prof.dr.ing. Paul Havinga, University of Twente
prof.dr. Wan Fokkink, VU Amsterdam

ISBN 123-45-6789-0123-4

Cover
by Sophia Langeraar

Inside Layout
by Nick Palmer

Printed by Ipskamp Drukkers BV.
http://www.ipskampdrukkers.nl

To my loving wife,
Thank you for always being,
Mi sol y mi amor

Acknowledgments

There is one person I have to thank for this amazing journey more than anyone else, and that is my promotor, Prof.dr.ir. Maarten van Steen. Without his confidence to take me on as a student to begin with, without the countless hours of advice, without the invaluable insights into what I was doing right (and wrong), I would never have completed this project. Maarten has a seemingly endless supply of energy and only minuscule sleep requirements. His tireless enthusiasm for his work rubs off on everyone around him. Though our schedules were rarely in sync (hence this research), it often worked to our advantage. I would code or write all night long, and send an update before I went to bed at 5 in the morning. A few minutes later, he would be awake and already reviewing what I had sent. Much like hot-bunking in the Navy, this can be extremely effective. Maarten, thank you for everything you have done for me in the past 7 years.

Thank you also to my co-promotor, Dr. Spyros Voulgaris. We've had endless entertaining debates and discussions on more topics than I can count. As a fellow deadline-worker, we put in some crazy hours, but (almost) always managed to pull it off. Your advice over the years has been consistently helpful, particularly your treatise on how to properly close your car door. Without your assistance I certainly could not have completed my research, and for that I am incredibly grateful. Thank you for being a great scientist, adviser and friend.

Thank you to my thesis committee members: Prof.dr. Wan Fokkink, Prof.dr.ing. Paul Havinga, Prof.dr. Konrad Iwanicki, and Prof.dr. Koen Langendoen. Your feedback on my thesis was critical, but constructive, and led to a much better final product. Thank you for the time and effort you have invested in my research.

To everyone, past and present, from the gossiping/wireless/sensor network group, thank you. Jan-Mark, you tricked me into starting this journey with fodder for an interesting Masters project. I always enjoyed our (rambling) chats and hope we get to work together in the future. Daniela, thank you for easing me into my project and helping me learn the ropes as a PhD student. Albana and Suhail, it was great to be able to talk through problems, work on the simulator, and perform experiments together. Best of luck finishing up your theses, and success in your next projects! Rena, your more theoretical approach to problems is a great counterpoint to my engineer's mindset, and I hope academia continues to treat you well. And Claudio, the latest addition to our group. Your first week on the job was helping to prepare and execute the largest experiment we performed thus far, but you took it all in stride. Always quick with a joke, and usually the last one to leave the office, thank you for your help over the last two years.

Caroline, you are a super-star. How you manage to keep our department running smoothly is beyond me. No matter how bizarre the request or problem you always have a smile and solution. Thank you for everything!

Thanks to all my other colleagues at the VU. You all made the VU an interesting and lively place to work. It is rare to enter the coffee room for a break and not fall into a discussion on the plot holes in the latest Star Trek movie, the merits of ACLs over Unix file permissions, or just a classic vim vs. emacs debate. You have made my time at the university entertaining and educational, and I will always appreciate that.

Thank you to all the good people at Chess. Frits, you've been a great source of ideas and inspiration, and more than a little bit of assistance. Siebren, Roland, Michiel, Kevin and Bob, you've all helped me along my way, and I wouldn't have made it through without you.

Thanks to Lex and everyone in the DevLab community. You have provided a forum for presenting and sharing new ideas, and have been invaluable in providing hardware for our larger experiments. Most of Chapter 3 would not have been possible without your help.

To all the friends I've made since coming to Amsterdam.

Marco, my first friend here in Amsterdam. We started off pursuing Masters degrees together as classmates, but quickly became friends. We told everyone from Day 1 we were doing Masters *and* PhDs, and in the end we both did. Thank you for everything, but especially for convincing me to stay in Amsterdam. Also, make an honest woman out of Angela. She's amazing, and everyone is doing it! (Hi Angela)

Nick, what a crazy ride, my good friend. At first I thought you were just the weird guy with the crazy mustache, but after 7 years I am 100% sure of it. We've been through it all in Amsterdam: births, deaths, weddings and everything in between. I'm so glad that you're off reaping the rewards of all your hard work back in the Golden State, and I wish you the best. Though you already have that, with your lovely wife Viola. Ciao ciao! Mille baci!! Viola, you always keep things interesting, even when you're napping. Thanks for all the parties and the "Yeahs!" The crazy part is, with your work ethic, I'm sure you'll have this cancer thing solved in no time!

Jeff, my other American Amigo. It wasn't for long enough, but we were three peas in a pod. It's always a blast spending time with you, making crazy plans for future companies or discussing the mechanics of wormholes. I sincerely wish all your plans work out, if for no other reason than you might let me hang out on your private island in 10 years. The main house will, of course, be architected by your soon-to-be-wife, Sophia. Soph, your energy is amazing and you love taking on new things. Thanks for being a friend, and especially for the KICK-ASS cover you made for me.

Eric & Pina, the first of this group to tie the knot. Eric, you're at the beginning your own PhD and are in for an interesting few years. Luckily you've found a wonderful girl to help you through it, Pina. Thank you both for all the game nights, whether it was Settlers of Catan, poker, or some other game I'd just acquired, you were always *in*. You two always look so happy together, I hope that never changes.

Alex & Kate, you crazy crime-causing duo! You're both fantastic and 1000% made for each other. Your recent wedding was "Da bomb," as the kids say. Alex, thanks for

fun times at high altitude and general debauchery at all altitudes. Kate, thanks for being the best hair-stylist a homeless lumberjack could ask for, and for never having the word "no" in your vocabulary.

To all the "Dutchies" that are always ready to prove all the stereotypes about Dutch people wrong: Hend and Annemarie, Tim, Livia, Tara, Jiska. You guys rawk! Whether snowboarding or crowdsurfing, making crazy hats or just playing Cowboys and Indians, you're always ready to have fun. I'll never forget some of the crazy times we've had, none of which should be explained here. And a special shout-out to Tim, for helping me with the translations of various summaries of my thesis. You're the man!

To my amazing parents and family, who always supported me no matter what.

Mom, you are the most amazing woman in the world. You taught me a love of knowledge, music and traveling. You kept us all together, showed us how to love one another unconditionally, and put up with a lot of crap. I love the fact that no matter how old you get, you're always an Xtra Zany Girl, ready to sing a song or dance a dance that you just made up. There aren't enough pages to say it fully: thank you.

Dad, I miss you every day. It breaks my heart that you won't get to see me finish this journey that I started while you were still here. You taught me acceptance, generosity, and a sense of spirituality that I treasure daily. I could always talk to you about anything, and knew you wouldn't be (too) upset if I called you at 3AM to talk about jazz. Who ever said life was fair?

Kevin, you were a father for us from day 1. Even though I didn't make it easy on you at the beginning, you always loved me as your son. It is an amazing stroke of luck that we found each other. Thank you for being with me for all of life's ups and downs, and I can't imagine a better man to head our family.

Jeremy, my little brother. Oh, how I abused you growing up. It seems like a lifetime ago that we were still under the same roof, arguing about anything we could find. It has been amazing to grow up with you, then to watch you scramble ahead of me in this race we call life. I cherish the time I have with you, and my glimpses into what you call "family life." You are an unbelievably good husband and father, so much so that I would consider it a victory if I can be half as good.

And last, but certainly not least, MariCarmen. You are the love of my life. Your smile lights up a room and blinds unsuspecting bystanders. Which can be a problem because you're always smiling. Your love of life and enthusiasm is infectious, especially when you aren't sleeping. You are brilliant and you are brave, though you would deny both. I am so proud to call you my wife. Thank you for being with me along every. single. step. on this crazy trip. TQMQPI!

The future's so bright, I gotta wear shades...

Contents

List of Figures

List of Tables

Part I Introduction

1. Introduction

Recent advances in electronics and embedded systems have made wireless devices become smaller, lighter, less intrusive, and significantly cheaper: a commodity. This enables the deployment of increasingly larger collections of such devices for a multitude of applications, mainly for the collection of observed data (sensor networks). There is no indication of a slow down in this trend. Quite on the contrary, we expect wireless networks consisting of tens of thousands of nodes to be common in the near future. As these devices continue to decline in size and cost, we anticipate a proliferance of *wearable* wireless devices. Whether physically worn on or as an item of clothing, such a device can spontaneously form networks with similar devices worn by people in the vicinity of the device. These networks will therefore consist of many mobile devices, or *nodes*, and possibly a number of fixed *routers* or *gateways*.

The potential uses of an ad hoc network composed of wearable wireless devices are vast. Consider, for instance, a (large) group of people at a conference or similar social event, each wearing a small unobtrusive electronic badge with a limited radio range. By simply measuring how often and for how long two badges are within range of each other, we can register social interaction and study the structure of the social network. Furthermore, by aggregating and disseminating data we can even stimulate social interaction, for instance by a social game where groups of people (e.g., students of the same department) increase their score by talking to members of other groups, and lose points when sticking among themselves. Finally, a family or group of friends attending a large social event may be informed when they come in close proximity to each other, helping them to stay in contact. Other applications easily come to mind, including group-based messaging, finding people with specific profiles, and crowd management, to name a few. Furthermore, these devices need not be worn by people at all, but at the right price-point could be attached and used to identify physical objects. The topology of the resulting network could be used to establish groups of items that "belong" together, later notifying the user when the node attached to one of the items leaves the neighborhood of the others.

In general we distinguish two operational modes of wireless devices: *externally* powered and *battery* powered. In an externally powered mode a wireless device receives its operational power from an external source (e.g., power outlet, generator), while in a battery powered mode a device relies solely on its battery for supplying energy. While some devices can operate in both modes (e.g., laptops, smartphones), we focus exclusively on battery powered operation in this thesis. When operating with an external power source concerns about energy savings are generally removed, and energy conservation is a primary objective of this work. Furthermore, any protocol suitable for battery-powered operation will function in an externally powered setting, but the opposite is not always true. In the networks described above the wearable nodes are battery powered devices, and the routers and/or gateways will typically be externally powered.

Of major concern to battery-powered wireless devices is their *lifetime duration*, and *energy* is the main factor determining it. Decreasing the energy footprint of

a wireless device boosts its lifetime in a reversely proportional way. *Lifetime predictability* is an equally important property of a wireless network for certain applications. Prolonging the network's life in a best effort manner is not enough for some applications that additionally require a reasonable estimation of the network's lifetime. To guarantee lifetime predictability, the use of energy should be constant and independent of operation-specific conditions, such as coincidentally high traffic or unexpected topology changes due to node mobility.

The requirements of *long* and *predictable* lifetime duration have led to the GMAC[1], or *Gossiping MAC*, family of protocols developed by Chess, B.V. In GMAC, nodes activate their radios to allow for message exchange for only brief synchronized periods, while during the majority of their operation the radio is powered off. GMAC functions in a completely *decentralized* manner, without the need of any coordinator nodes or infrastructure. GMAC also uses an epidemic-based communication model with periodic broadcast messages. GMAC forms the foundation for our solution and its operation will be described in detail in Chapter 2.

It is clear that to enable communication between two or more nodes, their active periods should be—at least partially—overlapping. In fact, to fully utilize the energy nodes spend on their radio circuits, their active periods should be synchronized as *accurately* as possible, to maximize the shared communication window. Synchronization of active periods in ad-hoc wireless networks is a nontrivial problem, notably due to the lack of a central coordinator and the inherently restrained nature of such devices. When confronted with the additional requirement of fixed-rate use of energy, it becomes a far more challenging problem, as solutions that asymmetrically put more burden either on the sender or the receiver, are ruled out.

The chief focus of our research is to develop a wireless network protocol for use by low-powered, energy-constrained devices in mobile ad hoc networks of potentially massive scale. In the next section, we will give a more detailed description of our research problem. In Sections 1.2 through 1.4, we provide a brief introduction to three important areas relevant to social ad hoc networks: ad hoc networking, low-power wireless protocols, and epidemic-based communication. We will then discuss our research goals in the context of these three topics in Section 1.5. Finally, in Section 1.7 we will present a survey of previous research relevant to our own work.

1.1 Research problem: enable social ad hoc networking

At the intersection of wearable computing, wireless ad hoc networks, and social networks, lies an area we have dubbed social ad hoc networks. Such networks are composed of nodes carried or worn by people, use wireless communication, do not rely on existing communication infrastructures, and are battery powered. The aim of these networks is to capture network dynamics at regular intervals. Social ad hoc networks can be composed of a large number of tiny nodes and will exhibit arbitrary mobility. We aim to enable the execution of distributed applications across large-scale social ad hoc networks. Potential applications for social ad hoc networks include:

[1] GMAC is protected by US Patent Application 12/215,040 and is available free of charge for academic use.

- *Registering social interaction*: By simply measuring how often and for how long two nodes are within range of one another, we can get an impression of the extent that two people are always in each other's company. This, in turn, will allow for analysis of the social structure of the network as a whole by simply letting each node report its observed neighborhood to a few monitoring nodes.
- *Stimulating social interaction*: Imagine that conference participants come from a limited number of non-overlapping groups (e.g., an inter-university event). By registering interaction between members of different groups, and subsequently disseminating and displaying information, we can stimulate social interaction between people who would normally tend to only interact with their own group members.
- *Monitoring group membership*: Consider a family or similar group of people attending a large social event. To keep track of each other's presence, each group member regularly broadcasts presence information. To prevent maliciously intended traffic analysis for tracking of a lost or isolated member, presence information is encrypted and flooded through the network to be recognized only by members of the same group, as in [1].

Our research problem is to design a wireless network protocol that will enable the deployment and utilization of large scale social ad hoc networks, as well as the execution of interesting applications on top of this network protocol. In order to implement a social ad hoc network, there are many challenges that must be overcome. We continue by looking at three different research areas that overlap in the setting of social ad hoc networks and that provide the context and inspiration for the solutions presented in the rest of this thesis.

1.2 Ad hoc networking

An ad hoc network is formed by a group of devices (nodes) that operate without the need of existing infrastructure in the form of routers or access points. Nodes in ad hoc networks communicate wirelessly. Typically networks are formed using radio communication, though other types of wireless communication (e.g., infra-red, acoustic) are possible. The first ad hoc networks were built in the 1970's, inspired by ALOHAnet [2]. Ad hoc networks are often *peer-to-peer* networks, but many exceptions exist. One example is the ad hoc mode offered by the 802.11 family of protocols, in which one node must act as a central *coordinator* for its neighborhood. Ad hoc networks can be grouped into three categories: mobile ad hoc networks, wireless mesh networks, and wireless sensor networks. To this trio, we add a fourth: social ad hoc networks.

1.2.1 Mobile ad hoc networks

A mobile ad hoc network is a self-configuring infrastructureless network of mobile devices connected by wireless. Because it is infrastructureless, all routing must be accomplished by the nodes participating in the network. Individual nodes are allowed to move, so the topology of these networks can change at any time.

example: Vehicular networks

1.2.2 Wireless mesh networks

A wireless mesh network is a communications network made up of radio nodes, characterized by organization in a mesh topology. Mesh topologies provide for many redundant paths, and reduce the likelihood of a node failure partitioning the network. Wireless mesh networks generally feature hierarchical routing via infrastructure devices such as routers and gateways. Typically, most of the nodes in such a network are stationary, though mobile nodes are often supported.

examples: 802.11s, 802.15.5

1.2.3 Wireless sensor networks

A wireless sensor network is composed of a number of wireless sensor nodes. These nodes are generally battery-powered, resource constrained devices, and are spatially distributed to cooperatively monitor physical or environmental conditions (e.g., temperature, humidity, ambient light level) in a given region. Typically these nodes coordinate in order to relay data through the network to a central server or gateway where the data can be further processed and/or acted upon.

Network protocols are often tightly coupled with the application or deployment involved. Constrained resources require simplicity and energy efficiency.

examples: Great Duck Island [3], Potato farming [4]

1.2.4 Social ad hoc networks

A social ad hoc network is an infrastructureless network of battery-powered wearable devices connected by wireless. This type of network exhibits aspects of the three previous categories.

First and foremost, the active nodes are mobile and we expect completely arbitrary mobility patterns. Because the nodes are worn by people, the network's topology will be dictated by the whims of the participants. This means that links between individual nodes are likely to be unstable, as a node may change its location at any time.

Second, even though the transmission range of these low-power wireless devices is generally short (in our experiments, on the order of 1-5m), we expect that node density will generally be high enough to create a mesh topology. That is, a node will often have multiple redundant multi-hop paths to other nodes in the network. These multiple paths increase the redundancy and reliability of the network's topology, but they also make the problem of scheduling collision-free access to the medium difficult.

Finally, the devices tend to be resource-limited due to their size. Because these devices are designed to be easily worn or carried, they must be small and lightweight. This rules out heavy large-capacity batteries, and heat-producing components like high-performance multi-core CPUs. As such, we focus on simple but efficient and scalable algorithms in this work.

examples: GMAC [5], Sociometer [6]

1.3 Low-power communication

The subject of power-consumption is an important one in the context of battery-powered devices. The lifetime duration of a battery-powered node is determined by its average power use over time. Simply put, a battery has a certain *capacity* and a device consumes energy at some rate until the battery's capacity is exhausted. A wireless device's energy consumption is attributable to a small number of factors. These are, in order of magnitude:

1. Communication
2. Computation
3. Non-communication I/O (e.g., sensor use, logging to flash)
4. Other (e.g., leakage current, battery wear)

The energy costs of the first two dominate the others, and hence the device's lifetime is determined by communication and, to a lesser extent, computation needs of the operating system and applications. An exception to this is the case of smartphones and laptops, where the use of the devices screen (i.e., non-communication I/O) often dominates total energy consumption. Though there has been some research into energy-efficient networking protocols for such devices (discussed below), the primary focus tends to be on bandwidth, latency and reliability. The majority of low-power networking research comes from the area of sensor networks, due to their composition of unattended (and sometimes inaccessible) battery-powered nodes. Thus, it comes as no surprise that most research aiming at prolonging the lifetime of wireless networks focuses on limiting the radio operation of the battery powered devices. Indeed, the radio circuit of some sensor devices are measured to consume three orders of magnitude more power than the rest of the hardware (CPU, memory, etc.), either when the radio is in transmitting or receiving mode.

Any time devices communicate via a shared medium, a Media Access Control (or *MAC*) protocol must be used. The MAC protocol regulates access to the shared medium, allowing communication to proceed with minimal interference. Many popular energy-aware MAC protocols use carrier-sensing or ready-to-send/clear-to-send handshakes (generally called CSMA, short for *carrier sense multiple access*), which necessarily use more energy than a properly synchronized TDMA (short for *time division multiple access*) protocol that avoids collisions without these mechanisms. Though we describe both types of protocols in this chapter, our research focuses on synchronized *probabilistic* TDMA algorithms.

Regardless of the media access method, the main way to limit the energy spent on communication is to limit the time for which the radio circuitry is switched on. This implies intermittently switching the radio on and off. The periods during which a node's radio is on or off are known as its *active period* and *inactive period*, respectively. The ratio of the time that a node's radio is on (T_{active}) versus the node's total operational time (T) is known as the *duty cycle*. That is,

$$duty\ cycle = \frac{T_{active}}{T} = \frac{T_{active}}{T_{active} + T_{inactive}}$$

For example, a node that is active for $100ms$ every second has a duty cycle of $\frac{100ms}{1000ms} = 10\%$ and will remain active about ten times longer than the same device operating without duty cycling.

1.3.1 Sensor networks

The sensor network community has contributed a wealth of networking protocols, all with a focus on energy-efficiency. For example, [7] describes 74 different MAC protocols designed for wireless sensor networks. Many of these protocols are tailored to specific use-cases, whether that be a particular application or communication class or even custom-made for a single deployment.

As is typical, a number of different trade-offs exist. Saving energy on communication often has costs in other areas. For example, the use of the duty cycling technique described above reduces energy consumption but adds latency to message exchanges when compared to an always-on protocol. Optimizing for power use generally forces concessions in the area of bandwidth and latency. As mentioned earlier, this is why energy-efficiency is not a primary focus of the 802.11 family of *WiFi* protocols.

Nevertheless, significant energy savings can be achieved by choosing (or designing) a MAC protocol appropriate for the deployment's use-case. Further savings can often be realized by also taking into account the communication pattern exhibited by the application. An example is the *extended preamble* technique used by some protocols where message broadcasts are infrequent. By requiring the sender to send a long preamble before its actual message, receivers can save energy by waking up periodically and listening to determine whether any neighbor has data to send. This technique imposes asymmetric costs on the sender and receiver, leveraging the fact that transmissions are rare.

1.3.2 Other examples

The the majority of research into low energy communication protocols is focused on the domain of wireless sensor networks, there are several exceptions. Among these are PAMAS, EC-MAC, and the low energy variant of the popular Bluetooth protocol, BLE.

In [8], the authors present the Power Aware Multi-Access protocol with Signaling, or PAMAS protocol. In this protocol, nodes utilize separate *data* and *control* channels. The control channel is reserved for RTS/CTS messages and a "busy tone" that is broadcast when a node has requested transmission. Using the control channel, the protocol is able to determine occasions when the node cannot receive packets (because a neighbor is sending a packet to a third node) or when the node cannot broadcast a packet (because a neighbor is receiving a message from a third node). In these cases the node's radio will consume energy for no utility, so PAMAS deactivates a node's radio in such situations. Their simulations indicate power savings of 50%

or more over an *always-on* version of PAMAS. However, the use of multiple channel frequencies is impractical on the nodes we use (see Chapter 3).

The authors of [9] and [10] describe the the Energy Conserving Medium Access Control protocol, or EC-MAC for short. EC-MAC avoids collisions by using an explicit reservation system, governed by a central *base station*. Nodes executing the EC-MAC protocol try to disable their radio during unused slots, and the base station attempts to schedule contiguous slots whenever possible in order to avoid unnecessary radio transitions. Their results indicate the power consumed by a 20 node network can be reduced by as much as a factor of 1000, compared to the more standard 802.11 MAC protocol. EC-MAC is also unique as it is a TDMA-based protocol. Nevertheless, the reliance on infrastructure (in this case, the base stations) makes it unsuitable for our purposes.

The Bluetooth low energy (BLE) protocol extends the well-established Bluetooth protocol to include optimizations for low-power use. BLE offers a data rate of 1 Mbps (equivalent to our hardware) and also utilizes low duty-cycle operation as the primary means of reducing energy usage. However, BLE is not ideal for our scenarios as it also uses a master/slave relationship between network nodes. This creates an asymmetric burden on the master nodes, which need to do more work than the slave nodes, and precludes our goal of a predictable device lifetime.

1.4 Epidemic communication

Epidemic communication is a term we use to describe a communication paradigm where information (messages) spread through a network similarly to the way that a virus infects a biological population [11]. The term *gossip* protocol is often used interchangeably with the term *epidemic* protocol, but here we use epidemic to represent the general class of algorithms and gossip to represent a specific sub-class of epidemic protocols.

In epidemic protocols it is typical to classify nodes into one of two states with respect to a data item, d_i: *susceptible* or *infected*. Infected nodes have a copy of d_i, while susceptible nodes do not. When a susceptible node receives a message, M, containing d_i, it becomes infected. An infected node that removes its local copy of d_i (e.g., its cache slot gets overwritten) returns to the susceptible state. Some models include a third state, *removed*, for nodes that cannot be reinfected after leaving the infected state. Finally, still other models include an *exposed* state, to represent susceptible nodes who have a temporary copy of d_i and can thus potentially infect other nodes but are not infected themselves.

The epidemic communication paradigm has seen use in a wide variety of areas. One of early use was in replication of databases [12]. Epidemic communication is popular in distributed systems research because of its fault tolerance (e.g., resilience to node failure), robustness (e.g., unaffected by topology changes) and scalability (i.e.,

with respect to network size). It is precisely for these reasons that GMAC utilizes this paradigm.

Epidemic protocols, and in particular gossiping protocols, were first investigated in the context of mathematical models of communication [13], but applications in sociology and real-life social networks soon followed [14]. In fact, gossiping protocols are modeled after people spreading a "juicy" rumor. This is yet another reason why we expect GMAC to be adaptable to our setting of social ad hoc networks.

We classify epidemic protocols into two broad categories, based on whether a node becomes infected by *all* data items in a received message, or by only a *subset* of them.

1.4.1 Flooding

In a flooding protocol, a node R rebroadcasts messages received from a neighboring node S in order that it might be received by other neighbors not within the transmission range of S. In these protocols, a message, M_S, is composed of a header, H_S, and a set of data items, D_S. Upon receiving M_S, node R will store the items in D_S in its local cache, C_R. Node R will subsequently broadcast a message, M_R, composed of H_R (an updated header) and D_S (the original set of data items). Hence, flooding protocols get their name from the fact that data flows through the network like water in a flood, eventually infecting all reachable nodes. As above, a node n is considered infected by item d if $d \in C_n$.

There also exist variants, such as local flooding protocols or probabilistic flooding protocols. In a local flooding protocol, messages are rebroadcast until a particular *hop count* is reached (limiting the propagation of data to only a local neighborhood of the original sender). In probabilistic flooding protocols, a message is rebroadcast with a certain probability, generally in an attempt to avoid overloading the underlying network with too much traffic.

1.4.2 Gossiping

A gossiping protocol is distinguished from a flooding protocol by the fact that nodes participating in a gossiping protocol only store and rebroadcast part of each received message. As above, an incoming message, M_S, contains a set of data items, D_S. In a gossip-based protocol, a node R receiving message M_S will select a *subset* $D'_S \subseteq D_S$ of the data items from the message to add to its local cache, C_R. Later on, node R will select a set of items $D_R \subseteq C_R$ to include in its own message, M_R. The manner in which a node selects these subsets is known as *selection criteria*, and determines how data items will propagate through the network. Examples of selection criteria include choosing to keep only the most recently generated items (to rapidly propagate new items and discard old ones), or only the least popular items (to prevent an underrepresented item *dying out*). Other important parameters are the size of a node's local cache, the number of items broadcast in each outgoing message and the number of items cached from each incoming message.

1.5 Research goals

The first, and primary concern for almost any mobile device, is *energy consumption*. As these mobile nodes are battery powered, careful and judicious use of a node's fixed energy budget is essential. The second problem we face is *scalability*. We require that our platform operates efficiently on a wide variety of network sizes, from a few tens of nodes (e.g., a birthday party or small restaurant) to thousands of nodes (e.g., a large sporting event or stadium rock concert). Our third major challenge is node *mobility*. In the social settings where our applications will be executing people are free to join and leave the network, and to move anywhere they please (i.e., make arbitrary changes in the network topology). These dynamic changes to network topology can wreak havoc on many algorithms (e.g., routing and leader election) which assume stable and symmetric connections between nodes. Another challenge is *adaptability*. In any network of such a large scale there will be variations in node density, radio interference and other environmental factors across the geographical region in which the network operates. Such factors must be compensated for in a local manner, that is, nodes must be made aware of their current environmental conditions and adapt to them. This leads to our final problem, *lifetime predictability*. Though different nodes in the same social ad hoc network may experience dramatically different local conditions, we require that they spend their limited energy budget at a constant rate. Without this feature, nodes in "interesting" areas (e.g., high density or mobility) will be more likely to exhaust their batteries and stop operating. Given these problems and challenges, our research goals are as follows:

Research goal 1: Low and fixed energy consumption Duty cycling is the primary mechanism for saving energy. One question is, **what is the minimal duty cycle that a node can use?** A node must periodically exchange messages with its neighbors in order to stay synchronized with them and compensate for its clock drift relative to those neighbors. This synchronization is a primary focus of our work and leads to another question: **to what degree must nodes be synchronized in order to communicate?** Low duty cycles also lead to high messaging latency, but this is mainly a potential issue for applications above the network layer that may have large bandwidth requirements.

Research goal 2: Highly scalable operation We target scenarios with at least one thousand nodes, but our goal is to scale to the order of ten thousand nodes. Networks of this size are much larger than typically seen in existing research, and the resource-constrained nature of the nodes we use make this level of scalability even more challenging. Computationally intensive or memory hungry algorithms must be ruled out. An important question is: **for what network sizes are our solutions effective?**

Research goal 3: Insensitivity to node mobility An interesting challenge of our research is that of node mobility. Much of the existing work in the field of sensor networks assumes that the participating nodes do not move. Many of the protocols that do allow mobile nodes do so by requiring coordinator nodes to control access within their local neighborhood. As we focus exclusively on decentralized operation and predictable network lifetime, such solutions are not appropriate as they place asymmetric costs on the coordinator nodes. This, in turn, increases the energy consumption of the coordinators, making them a potential point of failure for the network as a whole. We investigate questions such as **how does mobility affect our network protocols?** and **which types of mobility present the most difficulty?**

Research goal 4: Adaptability to changing network conditions Due primarily to the scale of the networks we investigate, we cannot assume that all nodes will have similar degree (number of neighbors) or reliability of their wireless links. Because such fundamental network parameters change based on the physical location within the network, nodes must be adaptable to variable network conditions. This is even more important when we factor in node mobility, as a node may begin in an area of low density and quickly move into a region of high density. We must ask, **can our solution quantify environmental conditions?** And, further, **how should our solutions react to changing environmental conditions?**

1.6 Contributions

As part of an overall solution, we promote the use of GMAC. GMAC is designed to run on highly constrained sensor nodes and it therefore has very low processing requirements and a small memory footprint. This serves to reduce energy consumption, but GMAC's chief mechanism for energy conservation is duty cycling. GMAC's use of periodic gossiping via broadcast combines naturally with the periodic nature of duty cycling.

The primary contribution of this thesis is an in-depth analysis of a variety of methods of synchronization, primarily for the purpose of efficient TDMA-based communication. In order that a participating node is always able to communicate with other nearby nodes, regardless of how or where they move, we must ensure all nodes share a common active period. Because we are interested in highly scalable solutions that do not suffer from central points of failure, algorithms involving coordinator nodes are rejected. In this work, we break down the problem of decentralized global synchronization into a number of smaller subproblems, and analyze each of them.

In Chapter 4, we evaluate different synchronization techniques in the context of static network topologies. In Chapter 5 we continue our evaluation, but focus on mobile network topologies. We look at a wide variety of network sizes and topologies in both instances. We examine a number of potential improvements to GMAC's existing synchronization policies, finding some that are quite effective and some that are less so.

In addition to evaluating new ideas in our simulator, we present a critical analysis of several real-world social ad hoc network deployments in Chapter 3. These deployments serve to validate our simulation results, although mainly in a qualitative manner. Despite difficulties encountered in our experiments, our results show that the techniques developed in our simulation environment are applicable to real networks.

Then, we present a simple but effective distributed application for estimating the number of nodes participating in a social ad hoc network in Chapter 6. Applications like this one are the foundation that adaptive protocols are built upon. Without a method of determining or estimating fundamental network parameters there can be no way for nodes to react to changes in those same parameters.

Our final contribution is an improved GMAC. With the additional functionality developed in the course of our investigation, GMAC does indeed enable the type of social ad hoc networks we envisioned at the beginning of this project. We have run, and continue to run, experiments involving hundreds of participants, each with a wearable sensor node. These nodes allow us to monitor a real-life social event, discover group membership among participants, and even offer feedback in real time.

1.7 Related work

Here we present discussion of and references to existing research in several areas that are relevant to the investigation described in this thesis. We address this work in two categories: wireless sensor networks and synchronization. As already mentioned, there is a large body of research on network-level communications protocols in the context of sensor networks. However, although a multitude of MAC layer protocols have been designed for a plethora of different target scenarios, to the best of our knowledge no work exists that exhaustively addresses all of the problems outlined in Section 1.5.

1.7.1 MAC protocols for sensor networks

In [15], the authors discuss why traditional synchronization mechanisms, like NTP, are unsuitable for ad hoc sensor networks. The cited reasons include lack of energy awareness, requirements of infrastructure (canonical time sources), and assumptions regarding connected operation and static network topology. The authors also present several principles for synchronization in ad hoc sensor networks, given different application requirements.

In [16] the authors give an overview of the problem of time synchronization in sensor networks and analyze several important protocols. They describe eight requirements of synchronization methods appropriate for sensor networks, and explain the

trade-offs involved in optimizing for one over another. Finally, they find that Reference Broadcast Synchronization (RBS) and Timing-Sync Protocol for Sensor Networks (TPSN) offer tight synchronization bounds, but TPSN is more energy efficient. Both protocols are discussed below.

The authors of [17] provide a more recent survey of time synchronization protocols designed for sensor networks. One of the focuses of their survey is the scalability of the examined protocols, however none of the protocols were evaluated on a network larger than 300 nodes. While this is reasonable for many application domains, it is much smaller than the social ad hoc networks we target.

In [18], Langendoen classifies WSN MACs into four categories of organization: random access, slotted access, frame-based access and hybrid access. Of these categories, only the slotted and frame-based inherently adopt the notion of duty cycles, with nodes alternating between active and sleeping states. In the following sections, we describe each of these categories.

Random access

The first category is that of random access. This is a class of protocols where nodes are free to access the medium any time. When a node has data to send, it simply sends it. These protocols trade energy efficiency for protocol simplicity, flexibility to topology changes, and robustness.

By and large, nodes periodically poll the medium, while senders use a long preamble to signal polling nodes to stay awake for the reception of a packet. These polling periods and preambles are designed to avoid collisions and reduce idle listening, and the length of the polling period determines the duty cycle. There are many refinements of the general 'long' preamble ('short' long preambles, 'strobed' preambles, even secondary 'wake-up' radios), but all assume pairwise (i.e., not broadcast) communication to reduce the preamble length. Although this kind of solution is apt for generally scarce, unpredictable traffic, it does not apply to our application domain, where all nodes are senders and they send at fixed intervals.

Examples in this category include Low-Power Listening/B-MAC [19] and WiseMAC [20].

Slotted access

By contrast, in slotted access protocols nodes agree on sleep/active pattern, wake up at the beginning of a slot, communicate, then return to sleep until the next slot. Also CSMA, but contention resolution is a big problem, generally results in using CTS/RTS which adds significant protocol overhead. Examples are S-MAC, T-MAC, SCP-MAC.

There are two MAC-level protocols of this class that are particularly relevant for our discussion. S-MAC [21], one of the main representatives of *slotted* access protocols, divides time in fixed-length slots of 1-3s and uses a 300ms active period, during which nodes compete for the channel using carrier-sensing to avoid collisions. T-MAC [22] improves upon S-MAC by adding adaptivity to traffic. Active nodes time out if they hear no traffic for 15ms, drastically reducing energy use in idle networks. A similar improvement, called 'adaptive listening' was also added to S-MAC.

In S-MAC, when a new node joins it listens for at least the duration of a whole slot to detect the presence of other nodes. If other nodes are present, it follows their schedule. Otherwise it picks an arbitrary schedule of its own. When multiple schedules are detected, a node follows them all, acting as a bridge between independently synchronized clusters (called "virtual clusters"). This, however, imposes on bridge nodes an energy cost that is a multiple of the cost for nodes following a single schedule, which is against our goal of fixed energy consumption and predictable lifetime.

Most importantly, both protocols ignore the fact that in the course of time, notably in large networks where maintaining synchronization across a long diameter is nontrivial, such a policy will eventually lead to the coexistence of a number of diverse schedules, multiplying the amount of energy used, while at the same time hindering the operation of broadcast-based communication protocols. Although this issue does not arise in small-diameter and short-lived networks, in networks of the size, longevity, and mobility we target at it constitutes a major shortcoming.

SCP-MAC [23] is a further optimization of the aforementioned protocols, lowering duty-cycles to as low as 0.3% by allowing channel polling at very short, scheduled intervals using a novel collision-avoidance mechanism during the active period. The reader should note that in SCP-MAC only one node may broadcast per duty cycle. This means that epidemic protocols will progress at much slower pace due to limited transmission bandwidth. Although SCP-MAC is significantly more sensitive to a tight synchronization than S-MAC and T-MAC, the issue of merging virtual clusters to a common schedule is overlooked in SCP-MAC, implicitly assuming a set of nodes that is and remains tightly synchronized.

Frame-based access

Frame-based access protocols combine *a number* of consecutive slots to form a frame, with each slot assigned to a node or pair of nodes. These schedule-based protocols operate in a way that is closer to TDMA. Keeping nodes synchronized with each other is even more crucial than in the case of slotted access protocols. GMAC belongs to this category. Other examples include TRAMA/FLAMA (Traffic Aware/Flow Aware MAC) and LMAC

In [24], the authors present two protocols: TRAMA and FLAMA. TRAMA/FLAMA has nodes regularly broadcast long-term information about data that flows through them and their neighbors. Using this and a distributed hash function, nodes create a collision-free transmit schedule. Packets include a bitmask representing the next

100 slots, and each bit represents whether or not the node intends to broadcast in that slot. FLAMA 'improves' upon TRAMA by using a pull mechanism to determine traffic flows. FLAMA includes special slots used for bootstrapping and node joins.

LMAC nodes broadcast a bitmask of their one-hop neighbors, allowing all nodes to compute (by ORing all masks) which slots are free. Nodes always broadcast at least a MAC header (possibly followed by data) in their slot, so other nodes know the slot is claimed. LMAC does not use ACKs, minimizing radio transitions between TX/RX. An improvement (Adaptive Information LMAC) allows nodes to claim multiple slots, reducing problems with slot-provisioning. However, LMAC assumes that a centralized gateway node will bootstrap synchronization, avoiding multiple clusters.

TDMA Time division multiple access, or TDMA, protocols form a sub-set of the frame based protocols. The chief distinction between TDMA protocols and general frame-based protocols is that TDMA protocols aim for a *collision-free* slot assignment. That is, an assignment of transmission slots to nodes that ensures two nodes whose transmission ranges overlap will not broadcast in the same slot. GMAC's basic slot assignment mechanism is probabilistic, so it does not technically fit in this category.

In [25], Cidon and Sidi propose an algorithm that allows a multi-hop network of N nodes to dynamically agree on a conflict-free TDMA schedule. However, it requires $O(N)$ slots per frame, which renders it inappropriate for scenarios involving thousands of nodes.

In [26], Arumugam and Kulkarni present an algorithm that deterministically establishes a TDMA schedule by a gateway node circulating a token. However, no attention is paid to joining clusters and keeping them synchronized, as nodes are assumed to be de facto synchronized. Additionally, the algorithm uses token circulation to establish an initial *coloring* for all nodes, limiting the scalability of the algorithm and rendering it unsuitable for mobile networks.

The same authors propose SS-TDMA [27], a self-stabilizing MAC protocol for sensor networks. It assigns slots deterministically based on (known) locations in a grid topology and is bootstrapped from a gateway node that also acts as a sink. The protocol is tailored to TDMA schedules for gossiping, however no duty-cycling or other energy-awareness is discussed. In addition, the requirement of *infrastructure* restricts the operational region to within the range of gateway nodes and makes the protocol impractical for ad hoc networks.

Hybrid access

Hybrid access protocols are defined as protocols that represent a combination of the above categories. They attempt to strike a middle-ground between highly organized frame-based protocols that require tight synchronization and random or slotted access protocols that are less organized and flexible.

The authors of [28] present ZebraMAC (Z-MAC), a protocol that attempts to combine the strengths of TDMA and CSMA in a single MAC protocol. Z-MAC essentially uses a TDMA-overlay on top of a CSMA MAC layer (namely, B-MAC) to improve efficiency. Z-MAC adapts its behavior depending on the congestion level in the network, and requires only loose synchronization between nodes. Z-MAC exhibits a better throughput per unit energy ratio than B-MAC under load, but performs slightly less efficiently than B-MAC in lightly loaded networks.

In [29], the authors present DESYNC, a biologically-inspired algorithm designed to interleave the active periods of nodes in a round-robin fashion. The authors also describe DESYNC-TDMA, a version of their algorithm designed for TDMA networks. An interesting approach, and although the authors suggest that DESYNC may work well in multi-hop topologies ([30]), it is unclear how the algorithm will work in mobile networks. The convergence time for single-hop networks is $O(N^2)$ rounds, where N is the number of nodes, so scenarios with high mobility may prevent the network from ever converging.

Crankshaft [31] and PMAC [32] schedule RX slots rather than TX slots, allowing nodes to wake up in only their scheduled receive slot rather than all receive slots. In both protocols, neighbors must share the same slot, and thus must contend within their slots for airtime. Crankshaft determines slots by the node's ID modulus the frame length. In PMAC, nodes announce their next n-sleep/1-awake schedule at the end of each frame. Between the two protocols, Crankshaft demonstrates superior energy efficiency, while PMAC exhibits better adaptation to network conditions.

1.7.2 Synchronization

There has been a significant amount of research in the area of synchronization between independent nodes participating in a common network. In this section, we will describe some of the most relevant research to our own work.

Clock synchronization

The issue of clock synchronization in the face of clock drifts is addressed by Tjoa et al. in SMART [33]. Although this paper is an inspiration for the clock synchronization algorithm adopted in GMAC, it does not deal with the orthogonal problem of joining clusters with non-overlapping schedules, neither does it consider duty cycling.

A number of papers address clock synchronization on sensor networks at the application layer, that is, the capability of nodes to communicate is orthogonal to their synchronization state. Consequently, duty cycling as well as detection and merging of different clusters are not applicable to such papers. Timing-sync Protocol for Sensor Networks (TPSN [34]) and Flooding Time Synchronization Protocol (FTSP [35]) rely on creating a spanning tree over the whole network, stemming from a globally elected *root* node. The cost of leader election and tree building make such solutions

unsuitable for high diameter and/or mobile networks. The cost of synchronization is confined in [36] by piggybacking synchronization information on existing application traffic and by proposing a completely distributed solution, free from the expenses of centralized coordination. Nevertheless, this solution assumes a reliable communication channel, which is unrealistic in general and particularly when duty cycling is in place. Reference Broadcast Synchronization (RBS [37]) provides reliable and accurate synchronization for low-power wireless devices. Furthermore, RBS has been implemented on a number of different hardware and radio platforms. An interesting question is how RBS would perform in the large-scale mobile networks we target. Participating nodes must maintain state (timing data) on broadcasting nodes, which is later exchanged with other nodes in the network.

Merging clusters (syncgroups)

In [38], Liu et al. describe a method for merging clusters in multi-hop 802.11 ad hoc networks, in contrast to the more common solution of bridging the clusters. Their method is based exclusively on the passive listening method (extensively described in Section 2.4.2). There are no details on the merge process itself, presumably nodes simply "jump" to their new schedule during the merge.

Mank et al. present Mobile LMAC in [39] and [40], removing assumptions about static topologies and using gateway nodes to bootstrap synchronization. The proposed merge protocol comes close to ours with respect to the part making a decision on which cluster to prevail. However, their evaluation is limited to networks of up to nine sensors, which is too limited to draw any conclusion with respect to scalability. Additionally, the Mobile LMAC protocol focuses on enabling nodes to achieve a high throughput channel even in the case of high network load, in contrast to to GMAC, which is designed for constant-rate gossiping between nodes.

1.8 Rest of the thesis

Though there are a multitude of MAC protocols for sensor networks and many different synchronization methods, we have found none that address all the challenges inherent in the social ad hoc networks that we propose. As such, the rest of this thesis describes our research towards enabling social ad hoc networking and the goals presented earlier. To conclude our introductory remarks, we now present a brief road-map for the rest of this thesis.

In part two, we discuss the software and hardware used throughout this thesis. In Chapter 2 we discuss the MyriaNed hardware platform that is the basis for all nodes we use in our experiments, describe the basic GMAC protocol, compare it with the standard OSI model, explain its methods of frame synchronization and slot allocation, and detail the API it presents to the application layer. In Chapter 3 we critically examine four of the many real-world experiments we have performed during the course of this research.

In part three, we evaluate the behavior of GMAC via simulation, with a focus on network synchronization and application performance. In Chapter 4 we explain the details of our simulation environment and present the results of our initial simulations investigating network-level synchronization in static (non-mobile) topologies. In Chapter 5 we extend this evaluation to include a diverse set of node mobility patterns, and present analysis of the performance of GMAC's synchronization mechanisms. In Chapter 6 we investigate the performance of a simple epidemic application in the face of network topology changes resulting from node failure.

Finally, in Chapter 7 we present our conclusions and some discussion about potential avenues for future research.

1

Part II Hardware and Software

2. GMAC: Gossiping MAC

In this chapter we will give a complete description of the operation of the GMAC (**G**ossiping **MAC**) protocol. An understanding of the way GMAC works is central to other concepts presented later in this thesis. We will begin by explaining the context and motivation that drove GMAC's development. We then present GMAC from the viewpoint of a traditional OSI network model, starting at the bottom. Finally, we present a few aspects that distinguish GMAC from other WSN MAC protocols.

Note that though it is certainly possible to adapt GMAC to other devices, much of its design was influenced by the hardware used by the nodes of the MyriaNed platform. As such, it is difficult to discuss the software and protocol (known as the *MyriaCore* and *GMAC*, respectively) without also taking into account the associated hardware. We will begin this chapter with a section describing MyriaNed hardware platform and the nodes that we use in our experiments.

2.1 Motivation

GMAC was created by Chess B.V. (`http://wsn.chess.nl`) for use in wireless sensor networks. Some of the initial goals for the GMAC protocol were a low and fixed rate of power consumption, decentralized operation, robustness to node failure (soft failure), scalability and adaptivity. As with virtually all sensor network software, minimizing resource usage (e.g., CPU, memory) is a crucial requirement. It is important to note that GMAC's original design is neither a contribution of this work, nor did it consider the presence of mobile nodes. One of our research goals is to add support of node mobility to GMAC's set of features.

GMAC was designed with the use of epidemic protocols (see [12]) in mind. Such protocols are generally resilient to the failure of individual nodes, enhancing the robustness of networks using GMAC. In addition, the broadcast-based nature of wireless communication is a good fit for gossiping or flooding protocols since every message sent can (potentially) be received by all other nodes within the transmission range. This is one important difference from similar protocols applied in wired networks: *neighbor selection*. In wired networks, a node can generally select neighbors with which to communicate uniformly at random from the set of all participating nodes. By contrast, in wireless networks a node cannot select its neighbors at all. A wireless node's neighborhood is determined by *physical proximity*, i.e., the set of nodes within the sender's transmission range.

2.2 MyriaNed nodes

Though we used three different models of MyriaNed nodes in two different capacities during the experiments described later in this thesis, there are many commonalities between these devices. We will begin this section with a discussion of the uses and similarities of these nodes, before moving on to discuss their individual deviations from the common platform.

In the course of our experiments, we use MyriaNed nodes for two distinct purposes. The primary use for these devices is as *active* nodes (sometimes called *badge* nodes) in the network. An active node is a full participant in the network, and broadcasts an application message each frame. An active node will regularly receive messages from other active nodes, and will use these message exchanges in order to maintain synchronization with the other active nodes and to spread application data throughout the network. The secondary use of the MyriaNed nodes is as *passive* nodes, or *sniffer* nodes. A passive node does not fully participate in the network and never broadcasts any messages. The passive nodes are used to observe, in real-time, the message traffic generated by the active nodes. Furthermore, a passive node does not operate independently, but rather as a USB device attached to a host, such as a personal computer or laptop. Thus, the passive nodes need not be concerned with power usage or storage constraints, merely timestamping each received message and sending the timestamp and message over the USB connection to the host PC. Furthermore, in the case that we want to give real-time feedback on the state of the network, the PC can simply forward the sniffed messages to a central *visualizer* machine. This machine can synthesize the incoming data from many sniffers and display it in a meaningful way.

Our experiments utilize three different types of MyriaNed nodes, sometimes called *MyriaNodes* for brevity. First and foremost, all three node types use the same Nordic nRF23L01+ radio chipset. This radio communicates at either 1 or 2 megabit per second (Mbps) in the unregulated 2.4GHz spectrum, sharing the same frequencies used by WiFi and Bluetooth devices. The Nordic chipset provides for a fixed-size 32-byte MAC packet, with additional physical-layer headers (including a 16-bit CRC) being added automatically by the radio itself. The nRF24L01+ does not have the ability to perform channel sensing or collision detection, nor does it provide any received signal-strength indicator (RSSI). This chip was chosen in spite of these limitations because it has extremely low idle power usage. Active nodes generally operate with the radio disabled for more than 95% of the time, so a high leakage current (the power consumed by the chip even when it is disabled) can significantly increase the overall power consumption by the device. In addition, all nodes use the same 32kHz real-time clock, or RTC. An external clock is required to wake the CPU up from deep sleep states, as its own internal clock is disabled while it is asleep. The ability to completely shut down the CPU during long idle periods is also essential to the low-power operation of the MyriaNed nodes. These clocks (actually, oscillators), like any clocks, have limited precision and will tend to *drift* in relation to another such clock over time. These devices carry a specification of $\pm 20 ppm$ (parts per million) from the factory. What this means is that in the time it takes clock X to count one million ticks

(about 30 seconds), clock Y could count as many as one million and forty ticks (if Y is as fast as allowed and X as slow as allowed) or as few as 999,960 ticks (vice versa). This variability in individual clocks is exactly what GMAC tries to compensate for with its synchronization mechanisms, described in Chapter 2.

2.2.1 Version 2

The MyriaNode Version 2 represents the previous generation of MyriaNed hardware. It is the simplest and least expensive node type of the three used in our experiments, and is no longer in production. The node uses an ATMega645 processor and has 4 kB SRAM. Though they have the same RTC as the other MyriaNed nodes, their CPU has only an 8-bit RTC register. This means that a V2 node must wake up the CPU every 256 clock ticks in order to increment a higher-level counter. If this wake-up is not performed on time, the counter register will roll over and ticks will be lost. These limitations reduce the potential power-savings from putting the CPU to sleep during long idle periods, and can also adversely affect a node's synchronization with other nodes. This makes the V2 nodes less than ideal candidates for our active, wearable nodes. Nevertheless, these nodes should perform adequately as passive sniffer nodes, whose function is described in Section 2.2.

The V2 nodes that we purchased came in weather-proof packaging with an attached USB cable for power and serial access. The included packaging increases the durability of the nodes and, together with the long USB cord, mean that we can freely place the V2 nodes around the experimental venue without worrying about spilled drinks, nearby power outlets, etc. The serial controller on the MyriaNed V2 provides for a maximum baudrate of 38400 *bps* (or 4800 bytes per second), so data could be lost if the incoming data exceeds the node's outgoing bandwidth and buffering capacity.

2.2.2 Version 3

The MyriaNode Version 3 is the most versatile MyriaNed node. Nodes of this type feature an Atmel ATMega128 processor with 8 kB of SRAM. This more modern processor offers a 16-bit RTC register versus the ATMega 645's 8-bit register. This means the V3 nodes can operate in deep-sleep mode longer and should be able to more accurately maintain their timing. The V3 nodes also include four colored LEDs, a Reed switch, an optional accelerometer, and an edge connector for attaching additional hardware devices (e.g., sensors, flash memory, or battery).

In our experiments, the V3 nodes are almost always used in conjunction with SED (**S**torage of **E**nergy and **D**ata) modules. The SED module can operate only when attached to a host V3 node via its edge connector, since the module has no microcontroller of its own. An SED module is composed of a $\frac{1}{2}$ AA battery, two 2 MiB flash memory chips, and an on-off switch. The addition of the SED module makes the V3 node functional as an active node, giving it self-contained power and stable storage for logging.

2.2.3 Chalcedony

The Chalcedony nodes are the latest addition to the MyriaNed family of nodes. This node type was designed specifically for use in our social ad hoc networking experiments. They were created with wearability in mind, and are meant to be completely self-contained. The main hardware used is identical to the V3 nodes described above, with the only differences being the physical layout of the components on the (much larger) PCB and the addition of several I/O devices. Specifically, the Chalcedony nodes include a black-and-white liquid-crystal display (LCD), a 4-way directional control with click input (similar to those found on first generation mobile phones), an on/off button, two 2 MiB flash memory chips, an ambient light sensor, an accelerometer, and a microphone. The nodes also include a connector for a large rechargeable battery pack, similar to those found in mobile phones. The LCD and input controls allow for significantly improved user interaction, by giving feedback to the wearer via the display as well as receiving instructions from the user via the 4-way input. The on-board flash memory provides space for the node to log important details during the execution of the experiment. Finally, the additional sensors (light, acceleration, sound) can potentially be used to distinguish exactly what the wearer is doing.

In light of their intended use as a wearable device, a protective case was designed simultaneously with the Chalcedony nodes themselves. This hard plastic case has cutouts to allow access to the attachment points for the neck lanyard, the 4-way directional input, the power button, the microphone, and the mini-USB connector. By leaving access to these crucial parts of the node open, the case must be removed only to program the node. Turning the node on and off, interacting with the node, and recharging the node can all be done with the case in place.

2.3 Comparison with OSI model

Though called a MAC protocol, GMAC actually incorporates aspects of the two layers of the traditional OSI model: the data link and network layers. This kind of cross-layer architecture is commonplace in sensor networks where resources are extremely constrained. By eliminating layers of abstraction, internal data structures can be reused and function call paths can be shortened resulting in reduced CPU and memory usage. In this section, we describe GMAC from bottom of the OSI model up. That is, we start with a discussion of the data link layer aspects and move up to the network layer functionality.

2.3.1 Data link layer

In the OSI model the MAC (or Medium Access Control) layer is a sub-layer of the data link layer. The MAC layer is responsible for allowing nodes to cooperatively access a shared medium, which in this case means the node's wireless radio. For two nodes to communicate, at minimum the following two conditions must hold:

1. The nodes must be within each other's transmission range
2. One node must have its radio in **transmit** mode, while the other must have its radio in **receive** mode.

In fulfilling the role of an OSI data link layer, GMAC tries to establish a common notion of time in order for nodes to share access to the wireless medium. Limiting the number of nodes that can transmit at the same time will reduce the likelihood of interference by overlapping signals and increase the chance that some nearby node (or nodes) will receive a message broadcast.

The other traditional functions of the data link layer are error detection and flow control. These features provide an essential foundation for the upper layers of the network stack to build upon. Specifically, without proper error detection an application cannot be confident of *data integrity*, and must either add such functionality at the application layer or risk unpredictable behavior resulting from corrupt messages. Without proper flow control, nodes that have data to send will be unable to access the medium in a timely manner.

Timing

At its most basic, GMAC is similar to a duty-cycled version of the slotted Aloha protocol [2][41]. As such, time is treated as a series of transmit/receive slots of a fixed duration. Unlike slotted Aloha, GMAC groups together a fixed number of slots into a frame. The purpose of these frames is two-fold: 1) to create a periodic opportunity for synchronization (a *heartbeat*) and 2) to create a structure for the application-level epidemic dissemination protocol, which generally proceeds in consecutive rounds. At the finest granularity, any hardware timer has a minimum atomic unit of measurable time, known as a clock *tick*. Each GMAC slot comprises a fixed number of clock ticks, dependent on the time it takes to send a packet and any *guard time* used.

Frames The largest practical unit of time that GMAC deals with is a frame. The length of a frame is a parameter chosen by the creator of the network. A frame is divided into an *active* period and an *inactive* period. Application-level communication (gossiping) happens during the active period, while MAC-level synchronization messages (known as **join** messages, described below) are exchanged during the inactive period. The ratio of the duration of the active period and the duration of the whole frame is known as the duty cycle and is represented by the symbol τ. We will refer to the number of active slots in a frame as S_A, the number of inactive slots as S_I, and the total number of slots in a frame as $S_F = S_A + S_I$. As an example, a frame with a duty cycle of 40% is depicted in Figure 2.1. Typical frame times used in our simulations and experiments are from 0.5s to 2s, and typical duty cycles are from 0.5% to 10%.

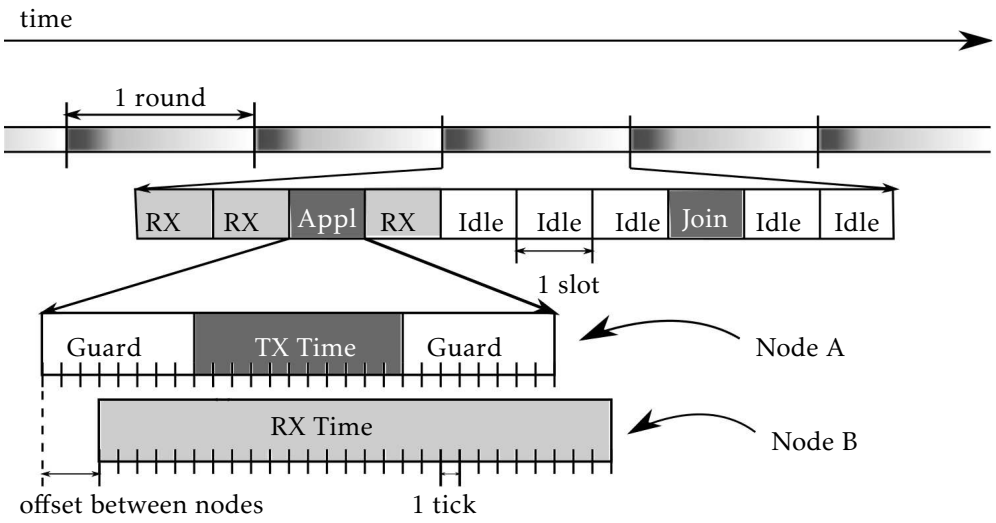


Figure 2.1: An example of GMAC's scheduling. We depict an abbreviated ten slot frame ($S_F = 10$) consisting of a four slot active period ($S_A = 4$) and six slot inactive period ($S_I = 6$, $\tau = \frac{4}{4+6} = 40\%$) for illustrative purposes.

Slots A frame is divided into a fixed number of slots. In each slot, a node executing GMAC can either broadcast a packet, listen for/receive a packet, or do neither (i.e., stay idle). Exactly which action the node takes in a given slot is determined by the node's *slot allocation strategy*, described in Section 2.5.3. The length of a slot is the sum of the transmit time and the guard time. In Figure 2.1, the guard time is depicted as a white region in Node A's slot diagram, while the transmit time is shown as the dark gray block in between. The transmit time is fixed and determined by the underlying radio hardware. The guard time, however, is a parameter that is determined at the time of compilation. A longer guard time should reduce the number of collisions in the network, but at the cost of increased energy usage due to the additional radio active time. This increased energy cost is incurred by all receivers, which must keep their radio active for the whole duration of each slot, indicated as the light gray block in the figure.

Ticks At a fundamental level the smallest unit of time that GMAC can use is a single clock tick. The duration of a clock tick, t, is the inverse of the frequency (f) of a node's timer, $t = 1/f$. If, for example, a node A determines that it is slightly desynchronized with its neighbor B, the measurement of the offset between the nodes will be accurate to at best t and will always be an integer multiple of t. The duration t also represents the smallest adjustment that a node can make to its timing. Continuing the example above, if node A calculates that it is 5 clock ticks ahead of B, it can add 5 clock ticks to the duration of its current frame. Thus, both nodes A and B should start the next frame at the same time, modulo a single clock tick. Both the transmit time (T_{tx}) and guard time (T_{guard}) are measured in clock ticks.

Error detection

GMAC provides for error detection by means of a simple 16-bit cyclic redundancy check, or *CRC*, performed over the full MAC payload. This CRC is executed by the radio hardware itself, as mentioned in Section 2.2. Generally an n-bit CRC applied to a data block of arbitrary length will detect any single error burst not longer than n bits and will detect a fraction $1 - 2^{-n}$ of all longer error bursts.

Ensuring data integrity typically involves sending *redundant* data in addition to the data that needs to be communicated. This redundant data can be used to check the validity of the message. For example, the CRC used by GMAC does not add any information, but contains only the remainder of the message data divided by a known polynomial. The problem is that this redundant data takes up precious space in GMAC's limited 40-byte physical packet. Nevertheless, data integrity must be provided with high probability to allow for proper functioning of the application.

Flow control

GMAC handles flow control very simply: any node that has data to send chooses a transmit slot during the active period *at random*. Though not always the case, it is assumed that a node's application will generate a message in each frame. In this way, GMAC establishes a maximum message send rate of one per frame. The maximum receive message rate is generally $S_A - 1$, the number of active slots in a frame minus one slot during which the node is in transmit mode.

GMAC thus has deterministic maximum send and receive rates, based on the number of nodes within transmit range of each other and the number of active slots, respectively. In neighborhoods where the number of nodes is much greater than the number of active slots, message collisions will be frequent. In areas of extremely high node density, communication may even break down entirely, due to all transmit slots being used by more than one sender. GMAC provides for limited adaptability to changing node density through the strategy module, as described in Section 2.5.3.

2.3.2 Network layer

The network layer of an OSI protocol stack is responsible for connecting networks and routing. In general, such functionality is an essential component in hierarchical network topologies in order for devices in different administrative domains to be able to exchange data. Because sensor networks tend to focus on measuring physical phenomena in a specific area, such functionality is less important in this domain. Nevertheless, sensor nodes generally need a way of getting data *out of* the local network and *into* the Internet or other networks. In addition, GMAC must establish a common packet format, so that a node knows how to interpret received messages.

Packets

The most important thing GMAC does is send and receive packets. A node executing the GMAC protocol is always in one of three radio states: transmit (*TX*), receive (*RX*), or *idle*. This is depicted in the middle row of Figure 2.1, showing an example slot allocation for a single ten-slot frame. As mentioned at the beginning of this section, ensuring there are neighboring nodes in the receive state when a node transmits a packet is GMAC's prime objective.

With the Nordic radio used by the nodes (see Section 2.2), only a fixed-size packet of 40 bytes can be sent. Eight bytes of the forty are used by the radio: a one-byte preamble, a five-byte address (used as a *domain*), and a two-byte cyclic redundancy check for data integrity. The remaining 32 bytes are free to be used by GMAC itself. GMAC's own header is two bytes, and contains only the slot (from the sender's perspective) in which the packet was broadcast. A sending node always broadcasts its message exactly T_{guard} ticks after the beginning of its transmit slot. Thus, using this recorded transmit slot, a receiving node can compute the timing offset between itself and the sending node.

The GMAC protocol distinguishes two types of messages: **application** messages and **join** messages.

Application messages GMAC nodes send at most one application message per frame, during the active period. The purpose of the application messages is twofold. The most obvious use of GMAC's application message is to communicate application data between nodes participating in a GMAC network. The second use is to maintain synchronization between these same nodes. The exchange of application messages offers a node the chance to compare timing information with its neighbors, and adjust accordingly. Throughout this thesis, we ensure that there is an application message for each node in every frame. That is, a node in the `SYNCHRONIZED` state (see Figure 2.3 below) will always broadcast a packet during its active period. This creates a steady flow of application messages and allows for proper synchronization. The rate of clock drift between a node and its neighbors determines how long it can stay synchronized in the absence of application messages.

Join messages Because of the low duty-cycles typically employed, GMAC networks can splinter into separately synchronized groups that are unaware of the existence of other nodes. We refer to these temporal isolated groups as **syncgroups** (discussed further below). GMAC nodes broadcast **join** messages during the inactive period. The purpose of these messages is to be received by a node from a different syncgroup during its active period. Since these messages are *only* broadcast during a node's inactive period, the reception of a **join** message during the active period indicates there are nodes nearby that are not synchronized with the receiver. The receiver can then adjust its own synchronization based on the timing data included in the **join** message.

Connecting networks

A primary objective of the GMAC protocol is to synchronize all physically connected (proximate) nodes into a single cohesive network, and thus does not have any functionality specifically related to bridging separate networks. The physical layer does support addressing, through the use of the domain field of the physical packet. This domain addressing allows multiple sensor networks to operate in the same physical area and using the same RF frequencies without experiencing *cross-talk*, or communication between the separate domains. In principle, a GMAC node could act as a *bridge* between multiple domains, but this has thus far been unnecessary. This is because typically networks separated into different domains serve different purposes (hence the separation), so routing messages between them is not needed.

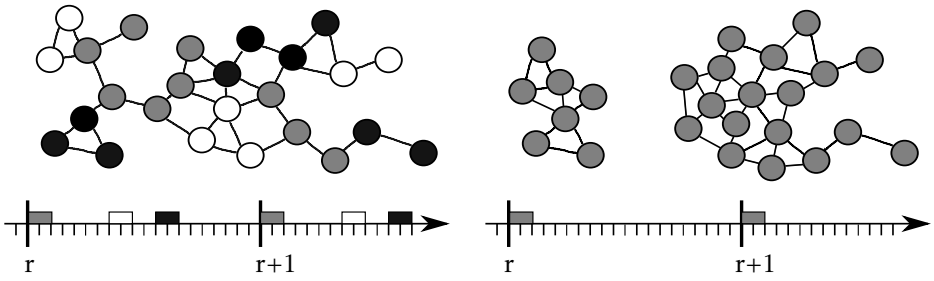
Routing

GMAC does not directly provide for one-to-one routing, but rather all-to-all via flooding or gossiping. This model, while not as powerful, is extremely resilient to node failure and allows for operation with far fewer resources (e.g., no routing tables) and a simpler API. Were routing to be required, it could be implemented at the application level as in [42].

In the event data needs to be routed from within the sensor network to external computers (e.g., for additional processing of data), this can be accomplished by using several passive nodes that simply receive any application messages from the nodes and forward them to a connected host computer. An example of this kind of operation is described in Chapter 3.

2.4 Synchronization

The duty cycle-based operation of the nodes makes synchronization of the active periods of their frames essential. If a node's frame is not synchronized with those of its neighbors, it is likely that its broadcasts will go unheard. Nodes whose active periods do not overlap cannot communicate with each other, effectively partitioning the whole network into disconnected clusters, called *syncgroups* (see Figure 2.2a). When considering mobile networks, the movement of the nodes may leave some of them temporarily isolated, or partitioned into physically separated groups called subnetworks, or *subnets*. Subnet disconnections are out of our control. Nevertheless, GMAC aims at a single syncgroup, so that when two subnets meet the nodes are readily able to communicate, as depicted in Figure 2.2b. Regardless of the type of message routing used, a node can receive a message from another node only if they are in the same subnet (they are direct or indirect neighbors) **and** in the same syncgroup (their active periods overlap).



(a) One subnet with three syncgroups (b) Two subnets with one syncgroup

Figure 2.2: The edges between the nodes represent wireless links. Nodes that are connected via (a chain of) wireless links are known as a *subnet*. The coloring of a node represents the group of nodes with which its active period overlaps, that is, its *syncgroup*. The axis at the bottom represents time, and is divided into individual slots. The numbered dividers demarcate frames of length T_{frame}. Finally, the shaded bars atop the time-axis represent the duration of the active periods of the similarly-shaded syncgroups.

The overall goal of GMAC's synchronization is to minimize the offset (see Figure 2.1), $\Delta T_{i,j} = T_i - T_j$ between all pairs i, j of communicating nodes. GMAC divides this offset into two parts: slot offset and phase offset. We define *slot offset* as the number of whole slots by which the two nodes differ, i.e. $\lfloor \Delta T_{i,j} / T_{slot} \rfloor$. We define *phase offset* as the number of clock ticks (intervals of t) the two nodes are set off within one slot. So, the total offset between the nodes is the sum of the slot offset and phase offset between them.

As implied above, groups of nodes whose active periods overlap are said to form a syncgroup. Synchronization of all nodes participating in the network then takes two forms: *maintaining existing synchronized groups* and *merging separate synchronized groups*. Because no two clocks are exactly identical, GMAC's synchronization module must maintain existing synchronized groups by compensating for the inherent clock drift between any two nodes. GMAC operates in a completely decentralized manner, so there is a chance that nodes will form independently synchronized groups or simply remain isolated. GMAC's **join** messages are responsible for joining these isolated nodes and subnetworks together.

2.4.1 Establishing and maintaining synchronized groups

For the purposes of this chapter, we have configured GMAC to use an active period of $N_{active} = 8$ slots, followed by an inactive period lasting until the end of a $T_{frame} = 1s$ frame. The reference hardware has a 32 kHz clock, so one timer tick is $t = 1s/32768 \approx 30\mu s$. The reference radio (see Section 2.2) takes about $300\mu s$ (10 ticks) to send a packet, and GMAC inserts a 9-tick guard time both before and after a transmission (see Figure 2.1). This gives a total slot time of $9 + 10 + 9 = 28$ ticks, or about $850\mu s$. Thus, on the reference platform, a frame has $N_{frame} = \lfloor T_{frame} \times 32768 \frac{tick}{s} / 28 \frac{tick}{slot} \rfloor$ slots. Using a $1s$ frame time we have $N_{frame} = 1170$ slots. With 8 active slots, we are left with $N_{inactive} = 1162$ slots in a frame, and a duty cycle of $\tau = \frac{8}{1170} = 0.68\%$. Since a node will have its radio active for only $\approx 7ms$ per second, it is crucial that

this active window is tightly synchronized with other nodes in order to maximize potential communication.

In this section we discuss how nodes first establish synchronized groups from a completely unsynchronized initial state, describing GMAC's operation in this start-up phase. We then explain how GMAC maintains synchronization between groups of nodes that have discovered each other and are operating in a synchronized state. Finally, we propose several improvements to GMAC's current maintenance algorithm.

Establishing synchronized groups

Nodes executing the GMAC protocol follow a simple finite state machine when initialized, shown in Figure 2.3. Nodes normally begin in the INITIAL_LISTEN state. In this state, a node will keep its radio active and continuously listen for a long initial frame with a random number of slots, $N_{frame} < N_{frame,0} \leq 2 \times N_{frame}$. If the node hears a message while in INITIAL_LISTEN, it will immediately deactivate its radio, calculate a timer adjustment to align the start of its next frame with that of the sender, move to the SYNCHRONIZED state, and then sleep until the computed start of its next frame. If, however, a node in INITIAL_LISTEN reaches the end of its frame without hearing a message, it will move to the SAY_HELLO state. In this state, a node will broadcast a specially tagged **hello** message in the first slot of the frame. After broadcasting the message, the node will switch its radio back to receive mode and enter the KEEP_LISTENING state. A node's behavior in KEEP_LISTENING is similar to that of a node in INITIAL_LISTEN. The differences are that a node in KEEP_LISTENING will maintain the standard number of slots in each frame, $N_{frame,i} = N_{frame}$, and will remain in that state indefinitely in the absence of receiving a message. A node in the KEEP_LISTENING state reacts to received messages in the same manner as described for INITIAL_LISTEN, aligning its next frame with the sender and transitioning to SYNCHRONIZED.

The astute reader will notice that in the above initialization behavior a node can spend a significant amount of energy in the INITIAL_LISTEN and KEEP_LISTENING states, as its radio will be active 100% of the time. This design decision runs contrary to GMAC's general philosophy of minimizing power use (i.e., radio active time), and was made for a number of reasons. Keeping the radio active eliminates the dependency of synchronization on internode communication, leaving only a dependency on physical proximity. That is, always listening allows the initializing node to (potentially) receive any message from an active node in its vicinity, regardless of *when* that message is sent. Furthermore, deploying a completely isolated node should be a rare occurrence. A wireless sensor node with no neighbors with which to communicate is almost completely useless. Thus, a fundamental assumption is that nodes will not start up in isolation. For these reasons, GMAC was designed to form an initial syncgroup with any neighboring nodes as quickly as possible.

Upon reaching the SYNCHRONIZED state, a node executes the normal active/inactive duty cycle behavior, as shown in Figure 2.1. A node selects a random slot in the active period as a transmission slot for an ***application*** *message* ("Appl") and treats

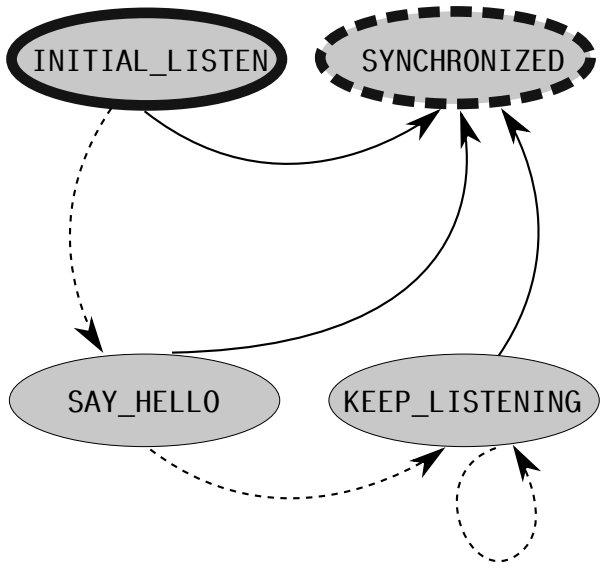


Figure 2.3: The finite state machine describing the operation of GMAC when a node starts up. INITIAL_LISTEN is the normal starting state, while SYNCHRONIZED is the goal state. A node will follow the dashed arrows if it receives no messages in a frame, and the solid arrows if it does receive a message.

the other active slots as receive (*RX*) slots. Each node will also select a random slot in the inactive period ("Join") in which to send a ***join*** *message*. A **join** message serves two purposes. The first is to aid in the merging of distinct synchronized groups, discussed at length in Section 2.4.2. The second use, since only the header is necessary for merging, is to carry debugging information about the status of the individual nodes. This provides a way of easily inspecting the internal state of nodes after they have been deployed. Each slot is composed of a number of *clock ticks*, and a single tick is the smallest adjustment a node can make to the length of its frame. By including the sender's slot number in every message, a receiving node can always determine the offset between its own notion of time and that of the sending node.

Because we evaluate network synchronization using a simulator, it is also possible to instruct the nodes to initialize directly into the SYNCHRONIZED state. By initializing some or all nodes at the same simulated time in the SYNCHRONIZED state, we can manually create syncgroups to evaluate network synchronization in a variety of scenarios.

Maintaining synchronized groups

Groups of nodes with overlapping active periods, that is, syncgroups, must constantly strive to maintain their synchronization. In the absence of any local corrections by the nodes, the tiny differences between their individual clock rates will eventually cause them to desynchronize. Nodes with faster clocks will continually start each frame a little earlier than the last, while nodes with slower clocks will start each

consecutive frame a bit later. Eventually the syncgroup will "drift" apart, the nodes' active periods will no longer overlap, and communication between them will cease to be possible.

We showed in [43] that GMAC's algorithm is capable of maintaining tight synchronization within connected networks of static grids of nodes. The method by which GMAC does this is called the *median* algorithm (see Algorithm 2.1), because it bases each timing adjustment on the median timing offset of all received messages during a frame.

Algorithm 2.1: The *Median* Algorithm

```
Median(numRxEntries > 0, array rxEntries, gain > 0)
    // sort received messages by offset (in ticks) from local time
    SortByOffset(rxEntries);
    // select the median entry
    medianEntry = rxEntries[numRxEntries/2];
    // scale the correction by gain = 0.5
    phaseError = medianEntry.tickOffset × gain;
    // adjust the length of the current frame based on phaseError
    AdjustFrameLength(phaseError);
```


This algorithm is admittedly quite simple, but in general performs well. It is important for scalability that the algorithm is based completely on *local decisions*. In each frame, a node will synchronize to the set of neighboring nodes from which it successfully received a message. By always selecting the median timing offset, GMAC ignores outlying data points and focuses on those in the center. The algorithm tries to force the fastest and slowest nodes in the network to match those closer to the median. Note that GMAC uses a gain of 0.5 to dampen adjustments and to prevent oscillations in the synchronization maintenance behavior.

It is worth noting here that GMAC was designed for static deployments on the scale of dozens to hundreds of nodes. As mentioned earlier, we are interested in networks at least an order of magnitude larger, so the sheer size of the networks we investigate may pose problems. Furthermore, although GMAC is designed to operate in a purely decentralized manner without any dependencies on specific peers, it remains to be seen how mobility will affect its operation. With the introduction of dynamic topologies, good network-wide synchronization becomes even more important. In a static network, a node will always have the same neighbors, so tight local synchronization with much looser global synchronization (that is, between nodes that are many hops apart) is perfectly acceptable. However, if a node can suddenly move away from its local synchronized group and position itself anywhere else, stronger network-wide synchronization will be required.

2.4.2 Merging synchronized groups

While generally GMAC has little difficulty maintaining synchronization within syncgroups where the active periods of nodes already overlap, we also must ensure that GMAC can merge together separate syncgroups to form a single, cohesive network. The situation illustrated in Fig. 2.2a, multiple syncgroups within a single subnet, significantly hinders the utility of the network as a whole. This is because internode communication is limited not only to the direct neighbors of a node, but also only to those neighbors with which it is synchronized. In order to achieve the possibility of, for example, multi-hop messaging, each node must discover other (groups of) nodes that are using a different active period, decide whether to merge with those nodes, and, if so, synchronize its own active period with theirs. When all syncgroups have been merged together and all nodes share a common active period, we say that the network has *converged*. Note that we reject solutions that attempt to "bridge" these separate active periods by requiring nodes in the overlap between groups to execute multiple active periods. Such behavior creates an asymmetric energy burden for those nodes that must run more than one active period per frame, jeopardizing our goal of a predictable network lifetime.

In our previous work, [44], we demonstrated that the default group merging behavior of GMAC was sufficient to achieve convergence for small networks, but that GMAC struggled to consistently converge larger networks. This problem of convergence can be broken down into three subproblems: *detection*, *decision* and *notification*, which we discuss in the following sections. At the end of each section, we also propose improvements to GMAC's current group merging mechanisms.

Detection

Before separately synchronized groups can be merged, they must first become aware of each other. We distinguish two methods of detection. In an **active** method, nodes *broadcast* a **join** message during the inactive portion of their frame, allowing other nodes using a different active period to detect the sending node's group. Note that **join** messages are always transmitted during the sender's inactive period, but can be received only during another node's active period. In a **passive** method, nodes *listen* during the inactive portion of their duty cycle to detect **application** messages from nodes in other syncgroups.

The effectiveness of active detection is mainly determined by the duty cycle of the network, τ. If $T_{active} > T_{inactive}$, then $\tau > 50\%$, so nodes are active for more than half of each frame. This implies that the active periods of all nodes must overlap to some degree, so separately synchronized groups cannot form. For this reason, we do not consider duty cycles greater than 50%. For duty cycles less than 50%, we can compute that the probability p_d of a detection event, that is, the probability that a message transmitted during one group's inactive period will be received during another group's active period (ignoring collisions), is equal to:

$$p_d = \frac{T_{active}}{T_{inactive}}$$

Based on the definition of the duty cycle, τ, we have:

$$\tau = \frac{T_{active}}{T_{active} + T_{inactive}} \Rightarrow T_{active} + T_{inactive} = \frac{T_{active}}{\tau} \Rightarrow T_{inactive} = \frac{(1-\tau) \times T_{active}}{\tau}$$

so that,

$$p_d = \frac{\tau}{1-\tau}$$

It is thus seen that the detection probability quickly becomes very low when τ is very small, which is exactly the case for the type of networks we are interested in. Nodes using a duty cycle of $\tau = 1\%$ can expect a detection probability of $p_d = 1.01\%$. As stated earlier, we use a duty cycle of 8 active slots out of 1170 slots in a frame, for a duty cycle of $\tau = \frac{8}{1170} = 0.684\%$. In this configuration, we can expect a detection probability of $p_d = \frac{T_{active}}{T_{inactive}} = \frac{8}{1162} = 0.688\%$ using active detection.

Passive detection offers a trade-off of increased energy consumption for faster detection. For example, a node could virtually guarantee detection of any other node in its range if it listened to the entire frame (e.g., a node in the `KEEP_LISTENING` state). However, this obviously defeats the original purpose of duty cycling the radio, and would rapidly deplete the node's battery. We could apply the duty cycling method to the passive listening by instructing nodes to listen to some percentage, p_l, of the inactive period, reducing energy consumption but also effectiveness. Note that this can be implemented as listening for an additional $p_l \times N_{inactive}$ slots every frame or by listening to the entire frame (an additional $N_{inactive}$ slots) with probability p_l. We chose to implement the latter method because listening to the entire inactive period eliminates the possibility that a node will fail to detect an unsynchronized neighbor due to listening to the wrong portion of its inactive period. Still, we will want to keep p_l as low as is practical, because the higher p_l is, the more energy is spent listening.

Active detection does have a decided advantage over passive detection. A whole *set* of neighboring nodes may detect the existence of another syncgroup at once, by a *single* message broadcast by one node of that group, provided the **join** message hits the active period of the neighbors. In the case of passive detection, each node would have to *individually* detect the presence of the foreign syncgroup, by paying the price of keeping its radio in receive mode during its inactive period. The disadvantage, however, of active detection is an increased chance of collisions, as the **join** messages sent from one group may collide with each other, or with **application** messages belonging to other groups. Both active and passive detection schemes will be heavily affected by the density of the network, in particular, the number of neighbors, or *degree*, of participating nodes. In addition, mobility will influence the effectiveness of both detection techniques as well. When a node n from one syncgroup detects a node m from another, there is a chance that, due to mobility, node m will no longer be in node n's range during the next frame, leaving n trying to merge into a syncgroup that has no members in its vicinity.

GMAC was designed to use active detection, with each node sending one **join** message during a random slot in its inactive period, as mentioned above.

Decision

Regardless of how detection happens, once a node from group B is aware of another group A, it must decide whether it should merge into A or if it should stay in B. Nodes cannot merge unconditionally, because otherwise the whole network may never converge as nodes merge back and forth between multiple groups. Our goal is to ensure that all the nodes converge into a single (possibly multi-hop) syncgroup, so that we should try to minimize the amount of time and energy spent on reaching a converged state.

The decision algorithm should implement a relation $\succ$ that provides a *total ordering* of the set of existing synchronized groups of nodes. That is, the decision relation $A \succ B$ determines whether group A is superior to group B. Thus, when a node in B receives a **join** message from a node in A, it should merge into group A if and only if $A \succ B$. The relation $\succ$ should provide the following three properties:

1. antisymmetric: if $A \succ B$ and $B \succ A$ then $A \equiv B$
2. total: $A \succ B$ or $B \succ A$
3. transitive: if $A \succ B$ and $B \succ C$ then $A \succ C$

In order to prevent cycles in the merging behavior, we propose enforcing a deterministic ordering of syncgroups. In static networks, nodes should eventually detect all other nodes/groups within their radio range. Provided that the network is connected, all nodes will eventually become aware of all other synchronized groups in the network. Because of the total ordering of these groups, nodes can always deterministically select the *best* group when making a merge decision. In theory, this should lead to network convergence as all nodes eventually merge into the best group.

GMAC was designed to use a heuristic mechanism to decide when a node should merge into a newly discovered group: if a received **join** message was sent during the first half of the sender's frame, then it is accepted as valid, otherwise it is discarded. Note that while sending **join** messages in the second half of a node's frame that will only be discarded seems wasteful, the messages contain debugging information so they are sent regardless of whether or not they will be considered valid. This timing-based relation, $\succ_t$, is meant to ensure antisymmetry (Property 1) for the decision relation. This is because, for any two groups, only one of them can send a **join** message in the first half of its inactive period that the other can receive during its active period. Furthermore, $\succ_t$ provides totality (Property 2), because the two groups cannot be desynchronized by more than half a frame, implying the active period of one group overlaps with the first half of the other's frame. However, $\succ_t$ does not provide transitivity (Property 3). If more than two groups exist in each other's range, there can be 'cycles', i.e., where nodes can merge from A to B to C and then back to A. In the best case, one (or more) of the groups in the cycle can be eliminated if the others can get all of their nodes to merge.

For example, if groups B and C could get all of the nodes in A to merge into their respective groups before A can get any nodes from C to merge into it, then the cycle would have been removed. In the worst case, these cycles can persist forever, leading

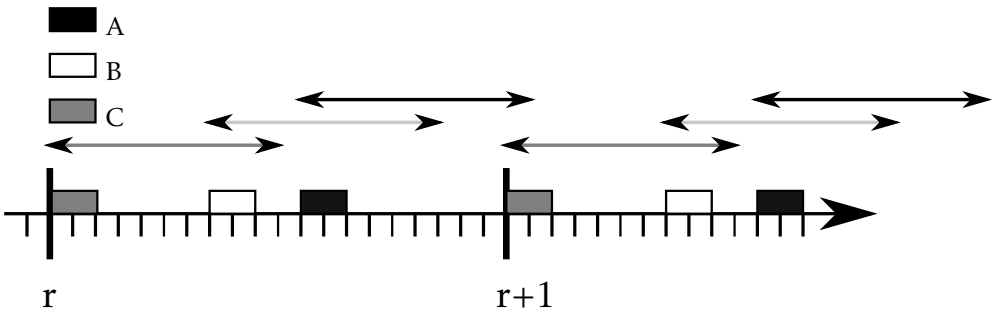


Figure 2.4: A graphical representation of the cycle problem in GMAC's decision mechanism. The axis at the bottom represents time in the same fashion as Fig 2.2. The additional shaded lines above the axis show the span of time where the **join** messages from the associated syncgroup will be respected.

to a network that never converges. A visual example of this effect is seen in Figure 2.4. Nodes in group A will accept **join** messages from group B, because the first half of the group B's frame overlaps with the active period of group A (i.e., $B \succ_t A$). Similarly, nodes in group B will react to **join** messages from group C. Finally, due to the timing of their active periods, nodes in group C will react to **join** messages only from nodes in A. It is thus seen that the three groups form a cycle, allowing for a node to merge from A to B in one frame, from B to C in a subsequent frame, only to later merge back into group A, ad infinitum. We will provide a solution to this problem below.

Note that taking mobility into account does not directly affect the logic of deciding whether or not to merge. Nevertheless, there are effects that we need to consider, specifically on the design of the decision relation.

Notification

Once a node has decided that it must merge into a new group, it should notify its own group of the merge. Though not strictly necessary, notification of the node's decision to switch from group B to group A can be rapidly propagated through group B (leveraging the group's existing synchronization), saving the need for repeated detections of group A. Because the probability of detection is proportional to the duty cycle, the networks we investigate will have very low detection probabilities. Propagating a notification of detections will reduce the number of detection events necessary to synchronize the entire network.

By default GMAC does not use any notification of discovered groups. Nodes that decide to change groups just silently merge. That is, they leave their old group by adjusting the length of their current frame to align their next frame with their new group. In situations with many groups, this can lead to isolated nodes as neighbors discover better groups and leave them behind.

Again, when taking mobility into account, we expect that the main effect on merge notification will be the same as for detection. Namely, that a node may receive a notification of a merge decision and adjust its next frame to synchronize with the new

group, only to find that the member(s) of that group are no longer within communication range. How drastic this effect is will depend on the density of the particular scenario as well as the speed of the nodes in question. When considering node density, the positive effect from the merge messages should be proportional to the density of the network. That is, the denser the network, the greater the number of nodes that can be notified of a merge detection/decision. We will examine both of these issues experimentally later in the thesis.

2.5 Other GMAC details

In this section we describe several unique aspects of GMAC that do not fit cleanly into the standard OSI networking model. GMAC provides for a somewhat modular structure of its source code, particularly with respect to the synchronization and strategy modules. We also discuss **join** messages, a method of allowing unsynchronized nodes to discover each other. Finally, we give a brief discussion of the API GMAC presents to upper layers.

2.5.1 Application layer API

GMAC's simplistic model lends itself to a simplistic API. The bulk of the interface consists of three function callbacks that *must* be implemented by the application:

1. **appInit** - called to initialize the application at boot-up
2. **appEvalRxMessage** - called upon reception of an application message, only minimal processing time permitted
3. **appPrepareTxMessage** - called at the end of an active period, allows deferred processing and generation of the next frame's outgoing message

Implementing these three functions is all that is required to implement an application on top of GMAC. Access to other devices (e.g., on-board sensors, LCD) is provided by the MyriaIoLib, and the details are specific to each individual node type (discussed in Section 2.2). In addition to the above functions, GMAC provides several optional callback functions. One example is **appLogFrameData**, which gives an application the opportunity to log relevant data on a per-frame basis.

2.5.2 Sync module

The sync module determines what synchronization adjustments to make based on the messages received during the current frame.

The standard synchronization algorithm used in GMAC is the *median* algorithm. This algorithm sorts all messages received during the active portion of the frame according to the relative difference between the sender's timestamp and the local node's own receiver timestamp. The median algorithm simply selects the median timing difference to synchronize with for this frame. The goal of this algorithm is to provide stability by ignoring outlier data.

2.5.3 Strategy module

The strategy module is responsible for dividing the frame into active and inactive periods, as well as determining which active slots will be TX, RX, or IDLE for each node.

Though GMAC provides several different strategy algorithms, the one that we use throughout this thesis is called *SimpleSlot*. This strategy uses a fixed number of active slots and random transmit slot selection. The default number of active slots is $S_A = 8$, but in principle any number $1 < S_A < S_F$ can be used. Random slot selection should work well in relatively sparse neighborhoods, where the probability of collision will be low. But in dense neighborhoods collisions will be commonplace and, if the density is high enough, communication may become nearly impossible.

The *DistributedSlot* strategy attempts to remedy this problem by adapting the number of active slots based on the perceived number of neighbors. We leave the investigation of this and other strategies to our future work.

3. Real-world experiments

In the course of our research, we have performed a number of real-world experiments. As our primary research goal is to design a set of networking protocols suitable for large-scale mobile wireless sensor networks, real-world testing and experimentation are essential. Here we will describe several of these experiments, demonstrating the correct operation of our network layer on actual hardware.

Because our research interest is in highly scalable protocols, we must necessarily test our algorithms with hundreds of nodes. Ideally we would be able to perform experiments with thousands of nodes, but the expense involved in purchasing such a large number of devices is prohibitive. As such, we have performed much of our investigation via simulation. Part of the challenge of performing real-world experiments in the domain of sensor networks lies in the effort involved in planning, preparing and executing experiments involving many individual wireless sensor nodes. Each device needs to be independently charged, programmed and tested. After the experiment, we must re-collect all the nodes, properly shut them down, and fetch the recorded log from each node individually. This process is extremely time consuming, and it cannot be automated.

This chapter will explain our methodologies, our hardware and software setups, and the outcomes of several of our experiments. We begin by describing some of the practical considerations that must be taken into account, we detail the types of measurements we take and how we capture them, and finally we discuss four important experiments in detail.

3.1 Practical considerations

While simulation allows us to almost effortlessly scale from 10 to 10,000 node deployments, the real world is much less forgiving. Practical constraints generally limit experiments to a few tens to maybe a few hundred nodes. For example, our largest deployment to-date involved approximately 220 nodes. Other difficulties encountered tend to be unforeseen and non-technical (see [45]), rather than bugs or algorithmic errors.

In this section we give details about some of the challenges encountered while performing, measuring, and analyzing social ad hoc networking experiments.

3.1.1 Manual labor

Even a small-scale wireless sensor network can require quite a lot of work. From programming and testing nodes to charging and transporting them, much of this work is manual labor and decidedly non-technical. It is often this type of manual labor that dominates the time spent preparing for, executing, and analyzing an experiment.

Many sensor nodes require a special device (e.g., a USB dongle) to program them, as is the case with the MyriaNed nodes. These *programmer* devices can quickly become a bottleneck when preparing nodes for a deployment. For example, our group has three MyriaNed USB programmers, limiting the amount of parallelization that can be achieved when programming nodes with a new software image. This USB programmer must also be manually connected to each device in series, eliminating any possibility of automating the process via scripts, etc.

Other time-sinks include charging and/or preparing the nodes' batteries. The SED modules (described above) used with the V3 nodes include a battery, but that battery is soldered onto the module. In order to change an exhausted battery, the old battery must be desoldered from the PCB (being careful not to damage any of the other tiny electric traces in the process), then a replacement battery must be soldered in its place. This procedure can easily take five to ten minutes per SED module, meaning that changing 100 batteries is easily a full day's labor.

3.1.2 Timing for measurements

Many synchronization algorithms aim to synchronize all network clocks with a single *master* clock, giving a clear point-of-reference. Our solution, however, is completely decentralized. This makes analyzing the network synchronization difficult, due to the lack of any central authoritative time source. In order to perform a quantitative analysis of a sensor network's synchronization, we will need timestamps that are accurate to within $10\mu s$ or so. This is because an individual clock tick on a 32 KHz clock is approximately $30\mu s$, and we need our measurements to be at least as fine-grained as a node's clock tick.

In a sensor network, each node has an independent clock and starts up at a unique time. As such, local timestamps from different nodes (i.e., a count of the number of clock ticks since boot-up) cannot be compared directly. Passive observation and recording of messages from active nodes adds yet another independent time source, that of the receiving sniffer node. Matching timing data between various sniffer nodes and active nodes can prove to be a headache, and is further complicated by the global time issues described immediately below.

3.1.3 Global frame number

Nodes executing the GMAC protocol attempt to maintain a consistent coarse-grained notion of time via a global frame number, as described in Chapter 2. Nodes will "jump forward" to match a higher frame number received from a neighbor's message. Corrupted packets or a misbehaving (byzantine or malicious) nodes can cause havoc with timekeeping by broadcasting high frame numbers. Since the frame number field is a fixed-width integer, upon reaching the maximal frame number a node's frame counter will "roll over" back to 0. These rollovers in the global frame number make matching packets received by the sniffer nodes with the proper entry from the sending node's log difficult, because there are often several entries with a given global frame number.

In order to resolve this issue, we have two main options which we call *roll down* and *roll up*. Using the roll down method, we assume the frame number in the logged message is valid, leading to sometimes having multiple logged messages from the same node during the "same" frame (which is not possible). The roll up method is more complicated, but should be more accurate as well. In this case, we look for the frame number, which is included in every message, to drop. That is, a message $M_{v,r}$ sent by node v during frame r followed by a message $M_{v,r'}$ where $r' < r$. A properly functioning node will never decrease its frame number, only add one to it or adopt a higher frame number from a neighbor's message. Thus, we can assume that node v's frame number rolled over between the time the sniffer received messages $M_{v,r}$ and $M_{v,r'}$ and we can add $2^{16} = 65536$ to the message's reported frame number.

3.1.4 Data integrity

In any networking scenario *data integrity* is important, but in the context of wireless networks it is even more so. What we mean by this is that the message M sent by node S should be received by another node, R, intact and unchanged. If instead R receives an altered, or *corrupted*, message M', the results can be unpredictable. Depending upon which part of the original message M was changed, the effects could range from the node crashing due to bad input to nothing at all.

Modern wireless networking protocols generally ensure data integrity by employing sophisticated data encoding schemes that add redundancy and, most importantly, error correction. These techniques are collectively known as forward error correction and work by encoding outgoing data into a format that uses more bits than the original. Hence, the decoding process on the receiving side can check for any errors in the data and even repair many errors without requiring retransmission. Such techniques are difficult to implement on the MyriaNed hardware. This is because the Nordic radios used utilize a fixed-size packet of only 32 bytes. Using forward error correction techniques would further reduce the useful size of an already small packet. Extremely efficient encoding techniques exist (like turbo codes and low-density parity-check codes), but require more resources than our nodes can provide. Even an efficient $(3, 2)$ code (that is, one that encodes 2 data bits into 3 output bits) would leave only 21 bytes usable of the entire 32-byte payload.

The Nordic radio used in the nodes provides for a 2-byte CRC on all outgoing messages. Nevertheless, CRCs are not perfect and in practice it is not uncommon for a corrupted packet to pass the hardware CRC check. This can occur more frequently in regions of high node density, due to the constructive/destructive interference from other nodes' messages. Because we did encounter problems with corrupted packets, we implemented an additional MAC-level 16-bit CRC check as well as a 20-bit "magic number", in order to better filter out incorrectly received messages. Because many of the fields in the packet header have no "wrong" value, we cannot generally distinguish correct packets from corrupt ones that happen to pass the CRC check. Thus, adding a magic number to the message header provides a simple verification that a known series of bits arrived completely intact and allows us to filter some additional corrupt messages.

3.1.5 Storage for logs

Both the Chalcedony nodes and the SED modules have only 4 MB of non-volatile flash memory for logging, so space usage is an important consideration. The full details of the active nodes' logging are described in the following section but, with one log entry written per frame, care must be paid that the minimum amount of data possible is logged. We typically use a frame length of 500ms, meaning that one hour of experimentation will result in the creation of 7200 log entries. Thus, logging more than 582 bytes per frame would fill the log in one hour. Since many of our experiments last for 6 hours or more, logging more than one hundred bytes per frame would be impossible.

3.2 Measurements

We take measurements from two different sources in our experiments. Active nodes log data about their operation and the messages they send and receive. Sniffer nodes observe messages sent by the active nodes, then timestamp and forward the messages to a controlling PC. The distinction between active and sniffer (passive) nodes was explained in Section 2.2.

3.2.1 Active node logs

At the beginning of a new frame, a node stores both its version of the current global frame number and the number of clock ticks that have elapsed since it began operating.

During the active period of a frame a node keeps a count of both application and **join** messages that it receives, as well as any invalid or ignored messages. As discussed in Chapter 2, a message is considered *invalid* if it fails either the MAC-level

Table 3.1: Logged data

Name	Type/Size	Description
timestamp	8-byte int	number of elapsed clock ticks (uptime)
frame number	2-byte int	local version of the global frame number
frame slots	2-byte int	final number of slots for this frame
frame events	2-byte int	bitmask of flags for important events
cluster tag ID	2-byte int	*is* portion of node's cluster tag
cluster tag epoch	1-byte int	*epoch* portion of node's cluster tag
received app msgs	1-byte int	count of application messages received
received **join** msgs	1-byte int	count of **join** messages received
invalid app msgs	1-byte int	count of invalid application messages received
invalid **join** msgs	1-byte int	count of invalid **join** messages received
ignored **join** msgs	1-byte int	count of ignored **join** messages
app data length	1-byte int	number of bytes application wishes to log
app data	VARIABLE	application data (unknown)
TOTAL	23+ bytes	

CRC check or the magic number check, and it will be *ignored* if it contains inferior synchronization information.

At the end of the active period the node records the total number of slots scheduled for this frame. After this GMAC calls the application's *appLogFrameData* function (see Section 2.5.1), in order to give the application a chance to make GMAC aware of any application data that should be logged alongside the MAC data for this frame. This will be discussed in detail in each relevant experiment section.

As can be seen in Table 3.1, the MAC data occupies 22 bytes of space while the application data requires $N+1$ bytes, where N is the number of bytes the application wants to log ($N < 256$). With a total of 4 MiB of flash memory for logging, a node can log a maximum of 182,361 frames (with $N = 0$). Any application-level logging will reduce this figure substantially.

3.2.2 Sniffer node logs

The passive sniffer nodes execute an always-on listening protocol, and simply record any and all messages they receive. The sniffer nodes will not capture every message broadcast within their range because collisions and other types of interference affect the sniffer nodes as well as the active nodes. As mentioned previously, the Nordic

Table 3.2: Packet data

Name	Type/Size	Description
transmit slot	12-bit int	slot index selected for transmission
magic number	20-bit int	constant magic number for error detection
mac-level CRC	2-byte int	cyclic redundancy check for error detection
cluster tag ID	2-byte int	sending node's cluster tag *id*
cluster tag epoch	1-byte int	sending node's cluster tag *id*
frame number	2-byte int	local version of the global frame number
merge tag ID	2-byte int	
merge tag epoch	1-byte int	*iff non-zero*: notification of superior cluster (*id*, *epoch*) with specified slot offset
merge offset	2-byte int	
app data	16 bytes	application data
TOTAL	32 bytes	

radios employed in these nodes do not support collision detection, so message collisions often appear the same as silence to a sniffer node. The sniffer nodes also do not check the MAC-level CRC or magic number discussed in Section 3.1.4. This has the fortunate side-effect that we can get some insight into how many corrupted packets exist in the network by checking the CRC and magic number logged by the sniffer.

In addition to the V2 nodes that we use as sniffers, we also employ a raw radio-frequency sniffer device, or *RF sniffer* for short. These devices use an FPGA and high-speed processor to monitor a specified frequency range. All radio-frequency data in this band is captured, and the device attempts to parse packets from the incoming stream of data. This is accomplished by matching the known header in GMAC packets, as well as checking each message's packet-level CRC to make sure the message is correct. The most important aspect of this device is its high-precision clock, which allows the device to timestamp received packets with a granularity of $1\mu s$ (about 30 times finer grained than the resolution of the clocks on the V2 & V3 nodes).

The basic packet format used by the active nodes is shown in Table 3.2. Each received packet is timestamped by the sniffer node before being inserted into a queue of packets to send to the host. Particularly for the V2 nodes (due to their slower CPU and lower serial bandwidth), a high rate of incoming messages could overflow the node's internal queue and lead to lost data or other problems. The sniffer node makes no attempt to validate or interpret the message itself, though it does verify the *physical*, or packet-level, CRC inserted by the Nordic radio itself. A sniffer node associates an 8-byte *local* timestamp with each received 32-byte packet. The interpretation of this timestamp differs depending on what type of node is acting as the sniffer.

In the case of an RF sniffer node, the local timestamp represents the number of $1\mu s$ clock ticks that have elapsed since start-up. The sniffer host runs software designed to timestamp, interpret and parse the raw radio frequency data and log the parsed packets and timestamp to disk. Much of this processing will eventually happen on the RF sniffer node's internal FPGA, as the software for this device matures.

If the sniffer node is a V2 or V3 device, the local timestamp is interpreted as the number of $\frac{1}{32768}s$ clock ticks that have elapsed since the node began operation. In this case, the sniffer host adds a *real-time* timestamp to the log, due to the relatively coarse-grained timestamping on the V2/V3 nodes. The host runs a simple program that simply continuously tries to read 40-byte records (8 bytes timing, 32 bytes packet) from the serial device, timestamps the reception of these records, and writes them together to disk. The sniffer hosts should all execute the network time protocol, or NTP, in order to keep their clocks as tightly synchronized as possible. Using the NTP-based real-time timestamps should allow us to match data received from different V2 sniffers, whether attached to the same controlling laptop or not. Note that there are variable-length delays associated with USB buses/devices, so the accuracy of the laptop's real-time timestamps may be limited by the timing jitter in the USB serial protocol.

3.3 Experiments

In the course of this research, our group has performed more than a dozen real-world experiments. The first experiments were of an exploratory nature: what could we do with a wearable network of wireless sensor nodes? We had grand aspirations, but our first experiment was marked by almost complete failure and we were forced to adjust our focus from the level of application development to that of (debugging at) the network level. In fact, it was not until our most recent experiments that we began to get most everything right and were again able to focus on application-level issues. Failure is often a far better teacher than success, and we learned from each of our mistakes. In this section, we will describe in detail four of the more interesting and significant experiments that we performed.

3.3.1 DevLab cafe

Our first social ad hoc networking experiment was designed around a simple premise: people (particularly researchers) tend to socialize with individuals they already know. We wanted to create an application that would take advantage of our Chalcedony nodes' ability to give users real-time feedback via the LCD display, and utilize this

Table 3.3: Experiment characteristics

Experiment	DevLab café	30-year celebration	The Big Game	ICT Open
Participants	50	220	36	110 (of 220)
Venue	DevLab, Eindhoven	Rode Hoed, Amsterdam	Intertain Lab, VU	WTC, Rotterdam
Event	Monthly mixer	Celebration	Pub-quiz game	Conference
Duration	2-3 hours	6 hours	3 hours	6 hours
Active Nodes	Chalcedony	V3 & Chalcedony	Chalcedony	V3 & Chalcedony
Sniffers	None	V2	V2 & RF	V2 & RF
Visualization	No	Yes	No	Yes
Video	No	Yes	Yes	No
Location Tracking	No	No	Yes	No
Application	*InCrowd*	*NeighborReport*	*NeighborReport*	*NeighborReport*

Table 3.4: DevLab Cafe experiment details

Purpose	Familiarize ourselves with GMAC API and MyriaNed nodes
	Test giving feedback to participants in a social environment
Active nodes:	
+Number/Type	60 Chalcedony
+Number of Active slots	8
+Transmit slot selection	Random
+Application	*InCrowd*
+Application data logging	None
+Frame number rollovers	Many
Sniffer nodes:	
+Number/Type	None
+Positioning/Coverage	N/A
Major issues	Synchronization, lack of experience
Observations	Nodes communicated with a seemingly random set of neighbors
	Nearby nodes were often unable to exchange messages
	Frame numbers were not always synchronized between nodes
Lessons learned	Start the nodes carefully to ensure initial synchronization
	GMAC's default merge behavior is insufficient for mobile nodes
	On node logging is necessary for data analysis

feedback to change the behavior of the individual participants. To that end we designed a simple game that we called *InCrowd*. In this game participants are assigned to a number of *teams*, and they gain points by interacting socially with people from a *different team*. Note that how participants are assigned to teams is up to the experimenter, but we generally used demographic information (i.e., university/company affiliation, research focus, academic status, etc). The details of *InCrowd* are elaborated below.

An interesting challenge is that it is difficult to determine what exactly constitutes a social interaction. Do we require participants to actually talk to each other, and if so, for how long? Is eye-contact or a wave from across the room enough to constitute a social interaction? What sensors could we use to measure such varied forms of interaction, and with what level of accuracy? Though not directly related to our research on network protocols, many such questions have been addressed in the literature (e.g., [46]) In the end, we elected to use *prolonged proximity* as a proxy for social interaction. That is, if node A receives messages from node B over N consecutive gossip rounds with no gaps of more than M rounds, then node A will be said to be socially interacting with node B. Though a very simplistic model of distinguishing social interactions, it is still non-trivial. For example, what values of N and M are appropriate? If M is chosen too small, many interactions may be missed due to random packet collisions or other transient errors. If N is too small, the algorithm will detect many false-positives, e.g., participants merely walking past each other or having separate interactions with their backs towards each other. Luckily, early small-scale experiments with the nodes revealed that the human body's absorption of 2.4GHz radio waves meant that a MyriaNed node worn on a person's chest could rarely communicate with a node directly behind it. This implies that a node is much more likely to receive messages from another node if their respective wearers are in close proximity *and* facing each other. If true, the node's wireless radio may be sufficient to detect social interactions without the assistance of other sensors.

This experiment would also serve as our first deployment with more than a small handful of nodes, and would thus teach us how much time-consuming work is involved in programming, testing, charging, transporting and deploying a social ad hoc network.

Active nodes

In this experiment, we used GMAC's default of 8 active slots and the *SimpleSlot* strategy with random transmit slot selection (described in Chapter 2). All messages were broadcast at the highest possible transmission power, in an attempt to ensure network connectivity.

The *InCrowd* application we designed works by having nodes keep track of which neighboring nodes they have received a message from for the last $N = 15$ frames. If a receiving node, R, receives a message from a particular sending node, S, for $N = 15$ consecutive frames with no gaps (consecutive frames without messages from S) of more than $M = 3$ frames, it records an interaction between itself and S and scores a point. A detailed description of *InCrowd* follows:

- All nodes maintain an array with one entry for each participating node. An array entry consists of three items: the frame numbers of the first and last time that node was heard from, and the most recent known score for that node.
- In every MAC frame, each node broadcasts a message containing its own MAC ID, its score, and up to 3 <MAC ID, score> pairs from its cache.
- For each application message received, the receiving node (R) will update its cached score for the sending node (S) and any additional nodes (A...C) included in the message. R will also check whether it can score any points from its message exchange with S. Note that because node scores are monotonically increasing, a higher score is always more recent.
- If it has been more than $M = 3$ frames since the last received message from S, R considers any previous interaction with S terminated and records the current frame number as the *start* time of a new potential interaction with S. Else, if R's *start* time for S was at least $N = 15$ frames ago, R scores a point and adds N to the *start* time for its interaction with S. Finally, the current frame number is recorded as the *end* time (last communication) for S.

Because the nodes maintain a cache of the best known score for every participating node, they can also make an estimate of each team's score by summing the entries for each node on the team. If the gossiping algorithm works well, all nodes will be able to maintain an accurate estimate of the total score for each team. By displaying the top 3 team IDs and their respective scores, we aim to motivate participants to socialize and improve their own team's score.

Setup

The event took place in the office of DevLab in Eindhoven, The Netherlands in May of 2009. The people attending this monthly meeting would be there to present their own projects using MyriaNed devices, as well as seeing demonstrations of what other groups were doing and, of course, *socializing*. The attendees came from a relatively small number of Dutch research institutes and companies, so we elected to use this information to divide participants into groups for the *InCrowd* application. We assigned each node a unique *MAC ID* in the range 1..64, and mapped node IDs to group IDs in the range 1..8 by $ID_{group} = ((ID_{node} - 1) \mod 8) + 1$. At the beginning of the experiment, we distributed nodes to participants based on a mapping of home organization to group ID. In this manner, participants would score points for socializing with people from other organizations, but not for interacting with people from their own organization.

While handing the nodes out to participants, we simply took a node from the correct group (modulus), turned it on, and helped them hang it around their neck. We did not give the nodes a chance to synchronize as a single connected network before distributing them amongst the attendees.

Results/Considerations

While we were very optimistic going into our first experiment, we quickly learned how much we did not know about how these networks operate in the real world. As discussed previously in Section 2.4.1, a node will initially synchronize to the sender of the first message it receives. Because we were incautious about activating and distributing the nodes and they were handed out simultaneously in different parts of the room, several syncgroups formed in the network. The default GMAC merge behavior (described in detail in Chapter 2) does not perform well in the presence of many syncgroups. The mechanism to make merge decisions does not guarantee a single syncgroup will be formed, and the result was complete chaos. Nodes would occasionally hear **join** messages from another syncgroup, and often ended up jumping from one syncgroup to the next as a result. The high transmission power used (meaning the entire network had a physical diameter of only 2-3 hops) and node mobility ensured that the network would never converge to a single active schedule.

The problems encountered with this experiment led us to reconsider our focus on application-level exploration to network-level fundamentals. The synchronization mechanisms used in GMAC would need to be improved to cope with large, dense mobile networks. Additionally, we would need to implement both MAC and application data logging in order to debug and analyze future experiments.

3.3.2 30 years of computer science in Amsterdam

Our primary goal for the experiment at the 30 Years of Computer Science in Amsterdam celebration on December 2, 2011 was to perform a very large-scale deployment. We had executed a number of experiments since the DevLab Cafe event, and felt prepared to tackle a bigger set of mobile devices. We prepared almost 300 nodes for the event, and approximately 220 of the total participants agreed to wear one of our badges. This occasion also marked the first time that we used any of the node's sensors, namely the accelerometer.

As a result of the previous failings of GMAC's network synchronization, we had spent a significant amount of effort investigating potential improvements within the OMNeT++ simulation environment (see Chapters 4 & 5. This experiment would serve as a test of whether the techniques we developed would prove effective in the real world as well.

Active nodes

As in previous experiments, we continued to use 8 active slots per frame and random transmit slot selection via the *SimpleSlot* strategy. In contrast to the DevLab experiment, here we used only the lowest available transmission power in an attempt to limit communication to only those nodes that were in close physical proximity. Additionally, the node software was updated to log important MAC and application

Table 3.5: 30 Years of Computer Science Celebration experiment details

Purpose	Larger scale than any previous experiments
	Verify efficacy of merge improvements from simulation
	Test capture/analysis of accelerometer data
Active nodes:	
+Number/Type	100 Chalcedony, 200 V3
+Number of Active slots	8
+Transmit slot selection	Random
+Application	*NeighborReport*
+Application data logging	Neighborhood, Accelerometer
+Frame number rollovers	Many
Sniffer nodes:	
+Number/Type	6 V2
+Positioning/Coverage	Entrance & back of hall, approx 35%
Major issues	Very high node density, logging space usage, handing out nodes
Observations	Fewer received messages than expected due to collisions
	Some logged messages appear corrupted
	Sniffer timestamps exhibit significant jitter
	Very long queue to initially hand out nodes & record IDs
	Many nodes completely filled their logs
Lessons learned	More active slots required for dense topologies
	Additional data integrity checks needed (e.g. CRC)
	V2 sniffer timestamps too variable for analysis
	Need more efficient way of distributing nodes to participants
	Restrict user logging to increase duration of logs
	Difficulty of synchronizing measurements (logs) across devices

data to non-volatile flash memory. These logs are essential in attempting to analyze a completed experiment.

For this experiment, we designed a new application, called *NeighborReport*:

- The application takes advantage of GMAC's gossiping protocol to share neighborhood data items throughout the network.
- Each neighborhood data item includes the creator's node ID (C), the frame number in which the neighbor nodes were observed (f), and up to three node IDs that were neighbors of C in frame f. Node IDs and frame numbers are both stored as 2-byte integers, so one neighborhood data item requires ten bytes.
- A node will include both a data item representing a sample of its own neighborhood for the previous frame and a recent data item from its cache.

The inclusion of a second data item serves to increase the speed at which data items spread through the network. It also allows the sniffers to gain insight into what is happening in parts of the network that aren't directly within their range. Data items created by nodes far away can be carried into the range of a sniffer either by node mobility or multi-hop routing.

This application was also designed to include the capture of accelerometer data from the sensor on Chalcedony nodes. This event marked our first use of any of the Chalcedony node's on-board sensors, and the aim was to use this data in identifying physical movements in the participants that are indicative of specific social behaviors. By comparing one node's accelerometer readings to those of its neighbors during the same frame, we hoped to identify social groups as well as distinguish group *leaders* and *followers*.

Setup

In this experiment, we mapped each participant's identity to a badge ID, and demographic data (e.g., university affiliation, role) were associated with the badge ID only (in an attempt to alleviate privacy concerns). The purpose of this was to see if we could later analyze the data to find any patterns in the recorded social interactions related to the chosen demographic data. The event would consist of a series of presentations in a large hall, interspersed with several short coffee breaks. Finally, the event was scheduled to end with a social mixer. We hoped that the socially enforced patterns of being seated quietly in rows during presentations followed by high mobility and mingling during the breaks and final portion of the event, would be evident in the logged data.

In preparation for the event, we placed V2 sniffer nodes (described above in 2.2) around the most high-traffic areas of the venue. The sniffers were located in the entrance/bar area and the back of the main hall. Because of the layout of the presentation hall in which the event took place, getting good coverage of the area was nearly impossible. The main hall had a 3-story ceiling, preventing us from hanging any sniffers above the seating area. Furthermore, there was an open balcony which

formed a second seating area. Being one floor higher and separated by columns and railings, those seated in this area would almost certainly be out of range of the nodes on the ground floor. If separated for long enough, these groups of nodes should drift apart, forming separate syncgroups. This would provide a demanding test for our protocols.

Our collaborators from the University of Amsterdam set up video cameras in one corner of the venue to be used for two purposes. First, they wished to use the video recordings in order to have ground truth for analyzing their detection of various social behaviors via accelerometer readings. The Chalcedony nodes (which were the only ones with the required sensor) sampled their 3-axis accelerometers at a frequency of 80Hz. They computed the mean of four consecutive 16-bit readings, and logged 20 of these averages per one second frame. This added an additional 120 bytes of log data (20 samples of three 16-bit integers) for each frame, making exhausting a node's data flash a realistic danger.

Results/Considerations

Overall, the experiment went quite well. One problem that we encountered was again a logistical one: distributing the nodes to participants. Because we wanted to capture demographic information about individual participants, we recorded the name and other details of each participant alongside the ID of the node they wore. Recording these details by hand took longer than expected, and caused a long queue at the registration table. In the future, we would need to improve the way we distribute the nodes to avoid such bottlenecks.

The nodes suffered from the rollover issue (described earlier) that plagued all of our experiments. Nevertheless, the timestamps from the sniffer logs allowed us to reconstruct much of the data set. Many nodes reported fewer neighbors than expected, which we attributed to the high node density and low number of active slots. This combination can easily lead to a large number of packet collisions, which (due to the lack of collision detection) appears to the active nodes as indistinguishable from not receiving a message. Another possible result of the high node density was an increased number of corrupted packets, which were identified in the sniffer logs.

Another problem was that the application logged too much data for each frame, which resulted in many of the nodes filling all of their available flash memory and therefore being unable to log any more data from that point forward. The limited space available necessitates a delicate balance between recording too much and risking data loss versus recording too little and not having enough data for proper analysis.

Additionally, this experiment revealed that we had problems resulting from a lack of *data integrity*. We found that, despite the CRC implemented by the radio hardware, corrupt packets still occasionally got through. After discussing this with other wireless researchers, it was brought to our attention that very high node densities can

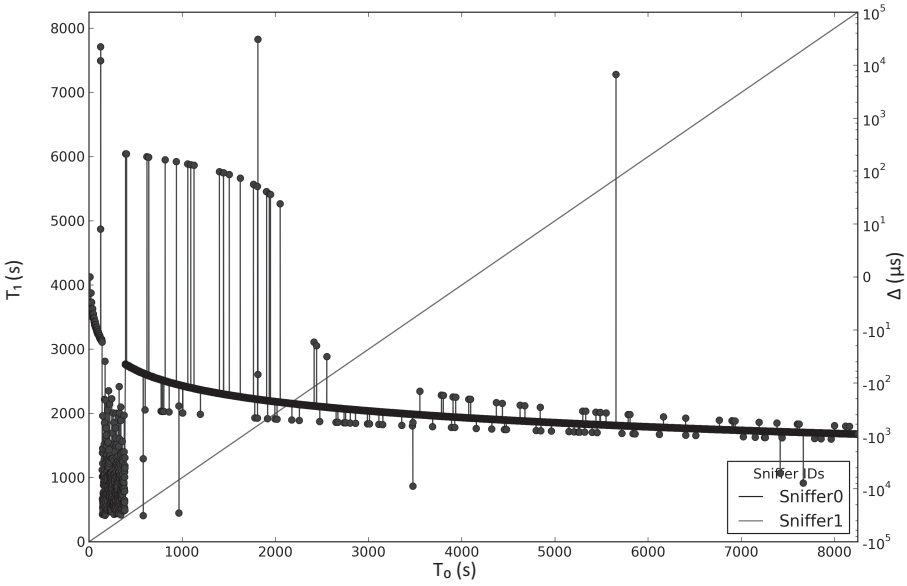
more easily lead to subtly corrupted packets that can still pass a CRC check (see Section 3.1.4). In order to combat this, we added a second MAC-level CRC check as well as a *magic number* into the message header. These two fields should greatly reduce the chances of corrupt messages being treated as correct messages.

One of the most important outcomes of this experiment was that we discovered that the V2 sniffers were not capable of timestamping data with sufficient precision to analyze the network's synchronization. In order to analyze the synchronization between the active nodes we need to be confident in the timing data returned by the sniffer nodes. This is because the active nodes are generally distributed over a large area, and hence beyond the range of a single sniffer node. In this case, we must be able to combine the logs from multiple sniffers in order to analyze the message timings across the whole network. Without stable and accurate timestamps from our sniffer nodes, merging sniffer logs together will be impossible.

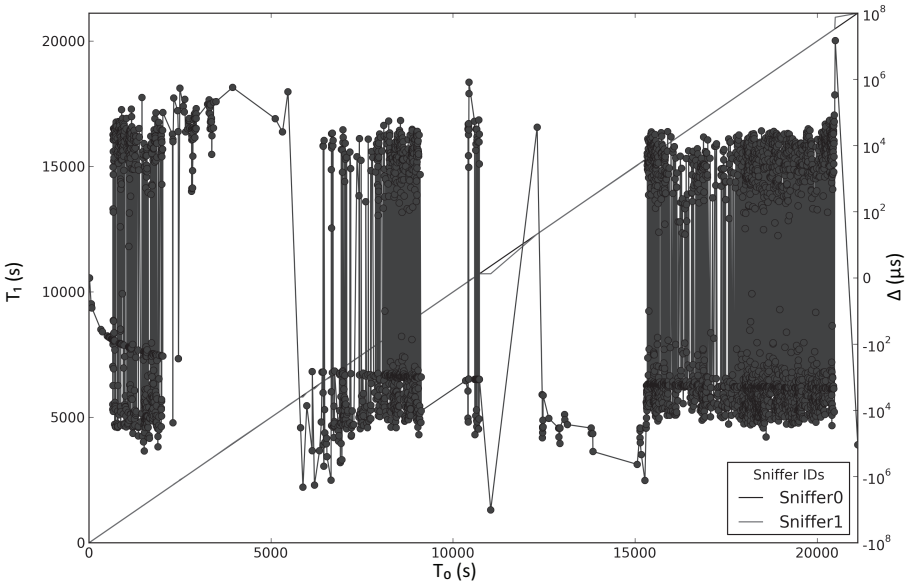
One way of analyzing the stability and accuracy of the logged sniffer timestamps is to compare the timestamps between two logs. That is, we search the logs of sniffer nodes S_i and S_j for a common message, $M_{v,r}$, sent by active node v during frame r. For every common message M received by both sniffer nodes, we compare the local timestamps from each sniffer, $t_M(i)$ and $t_M(j)$. While we expect there to be a difference between the local timestamps, $\Delta(M,i,j) = t_M(i) - t_M(j)$, we expect that difference to be constant. For example, if there is another common message M' received by both S_i and S_j, we would like $\Delta(M',i,j)$ to be very close to $\Delta(M,i,j)$. In Figures 3.1a and 3.1b we present the results of such comparisons. Each data point shows $Delta(M,0,1)$, that is, the time difference between the local time at S_0 and S_1 upon the reception of message M. We plot the local time at S_0 along the x-axis, the local time at S_1 along the left y-axis, and $\Delta(M,0,1)$ along the right y-axis. Local times (x and left y-axis) are shown in seconds, while the Δ is shown in microseconds.

In Figure 3.1a we show a "clean" data set, recorded using two sniffer nodes and two active nodes placed on a desk in an office at the university. While there were certainly potential sources of interference (e.g., WiFi access points, Bluetooth devices), this is a very clean data set. The active nodes and sniffers ran for about 150 minutes and the sniffers logged 13947 packets in common. In the figure, we can see that while the timing difference became more stable as the experiment progressed, it is certainly quite variable. This graph shows us that while generally the sniffers report consistent times for mutually received packets, there is *significant* variability in the timestamping. This is partially due to buffering issues on the sniffer nodes as well as latency in the USB serial protocol.

In Figure 3.1b shows a sample of the data recorded during the experiment. Here we again plot the timing of one sniffer node against another. The variability in the timing offset between the two sniffer nodes is even more pronounced here, because of the significantly higher network load (i.e., *many* more active messages) and contention for the USB bus from other sniffer nodes sharing the same host. Regardless of the cause, this level of timing jitter renders the V2 nodes an impractical tool with which to qualitatively measure the synchronization in the network.



(a) Clean desktop environment



(b) 30 Years of CS experiment environment

Figure 3.1: Sniffer timing comparisons

Table 3.6: The Big Game experiment details

Purpose	Distinguish types of social behavior via multi-modal sensing
	Control/record physical as much as possible for ground truth
	Capture accurate timestamps for analysis of synchronization
Active nodes:	
+Number/Type	36 Chalcedony
+Number of Active slots	64
+Transmit slot selection	Fixed
+Application	*NeighborReport*
+Application data logging	Neighborhood, Accelerometer
+Frame number rollovers	One
Sniffer nodes:	
+Number/Type	6 V2, 1 RF
+Positioning/Coverage	Uniform, nearly 100%
Major issues	Bug in timestamping packets received by RF sniffer
Observations	Nodes appeared synchronized and communicated with neighbors
	Incidence of frame number rollovers greatly reduced
	RF Sniffer log contained only packets, no timestamps
Lessons learned	Extra active slots improve neighbor detection in dense networks
	Additional message integrity checks mitigated packet corruption
	We did not perform adequate testing prior to deployment

3.3.3 The big game experiment

In this experiment, we again collaborated with faculty members from the UvA. To have greater control over the interactions of the participants, we designed a pub quiz-style game involving nine 4-person teams. By comparing the data from different parts of the experiment, we hoped to have an easier time establishing baselines for accurately discriminating social interactions from "casual" interactions caused only by proximity (e.g., someone simply standing near a group of people, but not interacting with them). The game was divided in a number of phases, some where we allowed the participants to freely mingle, some where we encouraged them to mingle with a specific purpose, and others where we forced them to remain in small groups. In addition, we used a number of other sensing devices (i.e., multi-modal sensing) to aid in distinguishing real social interactions from false/accidental ones. Just as in the 30 Years of CS experiment, we logged accelerometer data on the active Chalcedony

nodes. We also recorded video of the whole experiment, as well as audio using a separate system of wireless microphones.

The experiment is also notable because we affixed UbiSense nodes to each Chalcedony node. The *UbiSense* system uses a set of fixed anchor nodes to accurately determine the location (within centimeters) of the mobile "client" devices in a fixed area. The tiny client devices broadcast a message, and the anchor nodes compare the timing and angle of incoming messages in order to determine a precise location of the client node. This system should prove extremely useful in our data analysis, as it gives us an exact location for each node throughout the experiment. This would allow us to compare the detected social groups versus the actual position of each participant.

In addition to the UbiSense nodes for detailed localization, we also used several V2 sniffer nodes and our RF sniffer module to log application messages and timing details. The RF module, described in Section 3.2.2 should provide very fine-grained timing measurements, suitable for detailed analysis of the network synchronization.

Setup

This experiment took place in the Intertain Lab (http://www.cs.vu.nl/intertain/) at the VU. Participants arrived, were provided with drinks and encouraged to socialize with the other participants. Next, the players mingled in order to find compatible partners for the quiz later in the event. People were encouraged to seek out teammates that had strengths in quiz categories other than their own. After that, the attendees formed four-person teams for the remainder of the event. The quiz itself followed, and lasted approximately one hour. Finally, the event ended with a mixer where the participants were encouraged to further socialize and give us any feedback about the event.

The active nodes were running a modified version of the *NeighborReport* application described previously. The only difference is that due to the smaller number of participants, we replaced the list of up to 3 neighbor IDs in the data item with an 8-byte bitmask, with each set bit representing the reception of an application message from the corresponding node in that frame. This change made it possible for the sniffers to observe the network topology in even greater detail than previous experiments because nodes could now report *all* of their neighbors rather than just a sample. However, due to the additional space required for the extra MAC-level CRC and magic number, we no longer had enough space to send a cached data item as well as the node's own data item. Because we anticipated much better sniffer coverage (approximately 100%), we felt the trade-off of more detailed neighborhood reports versus the increased visibility offered by gossiping/flooding data times was justified.

Results/Considerations

From a social standpoint, the Big Game was a big success. All of the participants said they enjoyed themselves, and the top 3 teams won prizes for their efforts. We used significant prizes (iPods for first place) in order to motivate players to fully participate in the experiment. During the experiment, our observations indicated that network synchronization was performing adequately and nodes were able to communicate with their physical neighbors. Additionally, the nodes only experienced a single frame number rollover during this experiment. Whether this was due to improved message integrity because of the additional CRC and magic number headers, lower node density, increased number of active slots, an unknown factor or some combination of these, we cannot be sure. The sniffer logs include corrupted packets as before, but only because the sniffer nodes do not check the MAC-level CRC/magic number fields and simply record all received messages.

One of the biggest disappointments of this experiment was a bug in the RF sniffer program that resulted in the observed packets being logged, but without timestamp data. This, again, meant we were unable to make any quantitative analysis of the network synchronization. However, as in previous experiments, visual inspection of the nodes (via their on-board LEDs) in operation indicated that they were correctly synchronized and functioning normally. As we had determined the V2 sniffers were inadequate for measuring network synchronization, no attempt at analysis was made after the bug in the RF sniffer logging was discovered. This bug was a painful reminder that we must test *every conceivable aspect* of a deployment thoroughly before the experiment itself.

Despite this setback, the remaining data (accelerometer logs, node position, audio and video recordings) has proven useful for other researchers looking for different results. One example is a fellow researcher at the VU, who is developing metrics for distinguishing social interactions from a series of binary (yes/no) proximity data. That is, given a vector $v_{S,R}$ of Boolean values representing messages sent by S and received by R in consecutive frames (e.g., $T, F, F, F, T, T, T, F, T, T...$), can we accurately determine the beginning and duration of social interactions between the wearers of nodes S and R?

3.3.4 ICT open

In our last experiment, we pushed ourselves to try to improve upon our largest deployment yet. We again prepared almost three hundred nodes in hope of finding a large number of willing participants. However, turn out for the event was not as high as anticipated, and many of the participants were unwilling to wear one of our wireless badges. In spite of this, this experiment provided our best data set yet, due in no small part to the proper functioning of the RF sniffer logging. Attendees who did wear a badge were very interested to see and interact with our live visualization.

Table 3.7: ICT Open experiment details

Purpose	Perform large-scale experiment successfully and visibly
	Demonstrate our work to Dutch research community
	Capture accurate timestamps for analysis of synchronization
Active nodes:	
+Number/Type	100 Chalcedony, 200 V3
+Number of Active slots	64
+Transmit slot selection	Random
+Application	*NeighborReport*
+Application data logging	Neighborhood
+Frame number rollovers	Three or four
Sniffer nodes:	
+Number/Type	3 V2, 1 RF
+Positioning/Coverage	Located semi-centrally, approx 30% of main hall
Major issues	Lack of participation/volunteers
Observations	Attendees concerned about privacy implications of wearing node
	RF sniffer indicates network synchronization works correctly
	Frame number rollover problem still persists
Lessons learned	Full sniffer coverage essential to reconstructing experiment
	Improve message integrity & resistance to Byzantine failures
	Continue to address users' privacy concerns

Active nodes

The application run during this experiment was once again the *NeighborReport* application. The application was also modified for this experiment, this time to map 16-bit node IDs to 9-bit values. The purpose of this was to accommodate additional MAC header information necessitated by earlier problems with data integrity, as well as the need to use 16-bit node IDs with more than 256 potential active nodes. Using these "compressed" 9-bit node IDs, we were able to fit 14 node IDs (one for the sender and up to 13 neighbors) into 16 bytes.

Setup

In this experiment, the layout of the venue one again prohibited us from achieving full coverage with our sniffer nodes. The lobby of the World Trade Center is a wide,

open-air area with extremely high ceilings and very few locations to setup the sniffer nodes and laptops to run them. As such, we decided to focus our effort on the area immediately surrounding our demonstration. With some luck, our demo location was just to the side of the lunchtime eating area, affording us great sniffer data during the break periods.

As with most conferences, the day was broken up into presentation sessions interrupted by short break periods. This scenario provides a difficult test-case for any synchronization mechanism, since groups of nodes will be physically separated and unable to communicate for large portions of the day. During these periods of separation, groups of nodes in the same room will tend to stay synchronized with each other, but may drift apart from the other nodes in different rooms. When the participants (and the nodes) come together again during the break periods, the synchronization protocols must allow a node detect other unsynchronized nodes in its vicinity, decide whether to merge with them or to wait for them to merge with it, and finally to notify its own neighbors of any decision.

The most important difference to our previous experiments was the use (and proper function) of the RF sniffer device. This device allows us to take measurements of the timing of individual messages with an accuracy of about one microsecond. This is absolutely essential for any qualitative statements about the synchronization of nodes within the network. As discussed earlier, our other tools (namely the V2 sniffers and the node logs themselves) are incapable of the fine-grained timestamps required.

Results/Considerations

While disappointed that we were not able to convince a majority of the conference's attendees to wear one of our electronic badges, we still considered the experiment to be a success. We had about one hundred and twenty active nodes participating in the social ad hoc network, which was our second largest experiment to-date. In the future, we must try to find additional ways of ensuring the privacy of our participants and make it clear to them that we are doing everything we can to maintain their privacy.

With a correctly functioning RF sniffer, we were finally able to take high-precision measurements of the network synchronization. Each message contains the sending node's transmission slot number. Because a node broadcasts its message a fixed number of clock ticks from the start of its fixed-duration transmission slot, we can compute the exact time the sending node started that frame. We call this calculated time the node's *slot-0 time*. By comparing the slot-0 times of all nodes from which the sniffer logged a message on a per-frame basis, we can evaluate how the network synchronization proceeds throughout the experiment.

We present an example of the timing data recorded during the ICT Open experiment in Figure 3.2. Along the x-axis we show the "flattened" frame number using the roll up technique described in Section 3.1.3. That is, we group received messages

based on the (corrected) frame number in which they were sent. For each frame number, we compute statistics about the set of messages received during that frame. In the bottom plot of the four, we show the number of logged messages for each frame number. In the top three plots, we show the mean, median and standard deviation of the slot-0 times of the senders, with the y-axis representing time in seconds in all three plots. The lowest of these plots shows the mean slot-0 time, the middle plot shows the median slot-0 time, and the top plot shows the standard deviation of the slot-0 times for each frame.

These results are extremely positive and encouraging, demonstrating that synchronization is working properly. This confirmation of the accuracy of our many simulated experiments was sorely lacking from the project until this point. The results from this experiment indicate that nodes are generally synchronized to within a standard deviation of $< 10ms$, or about 12 GMAC slots. Since we used an active period of 64 slots for this experiment, this strongly indicates that the active periods of all nodes overlap to at least some degree, which in turn means communication amongst all participating nodes should be possible. This was one of the primary goals of our research, and it is extremely rewarding to have confirmation of its efficacy.

While these techniques have been shown effective on the GMAC protocol, they should apply to any low duty-cycle MAC protocol. For example, any algorithm that maintains multiple "virtual clusters" as described in Section 1.7 could instead use the cluster merging methods that we present in the next chapter. Our synchronization algorithms do not require specialized hardware or unrealistically accurate clocks. If anything, our solutions could be considered minimalistic, as they do not require a radio with support for collision detection or received signal strength indicators.

In future experiments we must strive to have 100% V2 sniffer coverage, as well as investing in additional RF sniffer modules. The fine-grained timestamps recorded by this device are essential for confirming or disconfirming the verisimilitude of our simulations.

Finally, we must continue to investigate new and existing methods of ensuring the integrity of messages sent by the active nodes. As it stands, a single malfunctioning node could cause havoc to an otherwise properly synchronized network, while a malicious node could easily impair its function altogether.

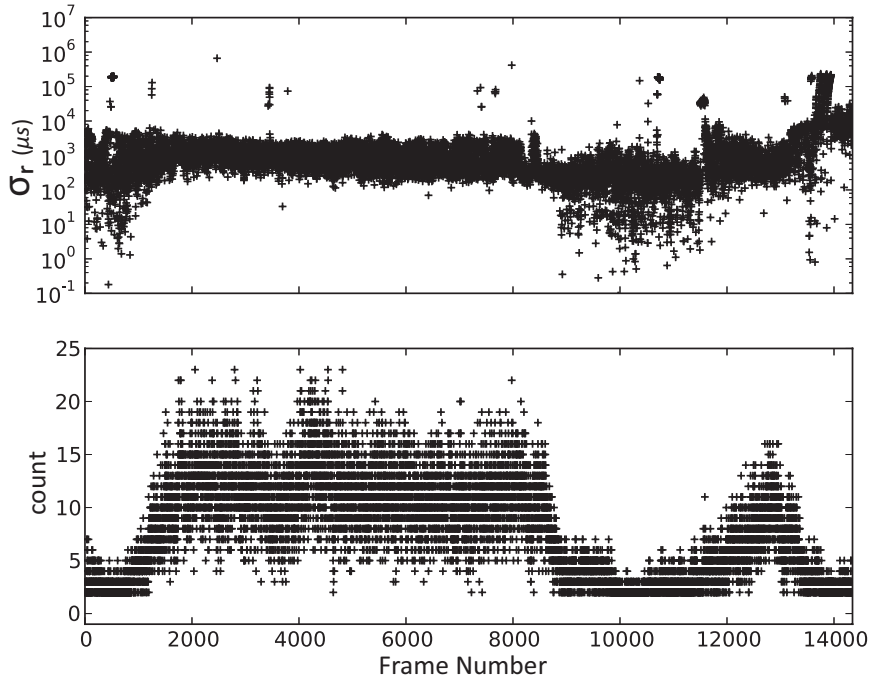


Figure 3.2: Timing results from the ICT Open experiment

3

Part III Simulating Wireless Ad Hoc Networks

4. Synchronization in static network topologies

Although this work is intended to be deployed for very large networks, we can only expect adoption if we have first explored the design space of our solutions. Simulation is an excellent tool to do this. Furthermore, we need to deal with the fact that the cost and effort of hardware development for real-world deployment is considerable. For example, solutions will most likely have to be embedded in an attractive way, such as bracelets or watches. Only now, as we are wrapping up this research, are we witnessing platforms that could be used for this purpose, including advance smartphones and programmable watches. We have therefore deliberately concentrated on large-scale simulations in combination with experiments involving up to several hundreds of (unattractive) sensor nodes. The remainder of this thesis focuses on experimentation through simulation.

As explained in Chapters 2 and 3, the conservation of energy is of paramount importance in such resource constrained environments, and this is generally achieved by limiting the time for which the radio circuitry is switched on. The implied intermittent radio on and radio off periods lead to the notion of a duty cycle. This duty-cycled operation of GMAC leads to the requirement of node synchronization. It is clear that to enable communication between two or more nodes, their active periods should be—at least partially—overlapping. In fact, to fully utilize the energy nodes spend on their radio circuits, their active periods should be synchronized as *accurately* as possible, to maximize the shared communication window.

The synchronization of active periods in ad-hoc wireless networks is a nontrivial problem, notably due to the lack of a central coordinator and the inherently restrained nature of such devices, as discussed in Section 2.4. In addition, we are restricted by GMAC's design goal of a predictable node lifetime (i.e., fixed-rate use of energy). This makes it a far more challenging problem, as solutions that asymmetrically put more burden either on the sender or the receiver are ruled out.

It is precisely this network-level synchronization that we investigate in the course of this chapter. Here focus on *static* network topologies, and evaluate of mobile scenarios in the next chapter. Note that although our discussion is presented in the context of GMAC, the methodologies, principles, and algorithms we propose can be generalized to virtually any MAC protocol with a very low duty cycle.

In this chapter we will begin by giving a detailed description of our simulation environment. This environment is used throughout the rest of this thesis to evaluate different aspects of GMAC's network-level synchronization and application-level behavior. The contributions of this chapter, improvements to the default GMAC synchronization behavior described in Section 2.4, are discussed in Section 4.2. Then in Section 4.3, we describe the specific simulation setup and parameters used to generate the results in this chapter. We present the simulation results of a number of different network topologies and activity scenarios in Section 4.4, before finally presenting our conclusions in Section 4.5.

4.1 Simulation environment

In this and the following two chapters, we will evaluate various aspects of network synchronization and application behavior through the use of simulation. In this section, we will give an overview of the setup and operation of the simulation environment used to generate the results presented throughout this thesis.

4.1.1 Simulator

OMNeT++[1] ([47], [48]) is an open-source discrete event simulator, and the MiXiM extensions provide a framework for wireless and mobile networking simulations. We conduct our simulations using the MiXiM extensions[2] ([49]). The OMNET++ platform is expressive, efficient, modular, and increasingly the de facto simulation environment for mobile ad-hoc and sensor networks, while MiXiM provides support for mobility and wireless network protocols.

Nodes

We have used OMNeT++ to extend the base MiXiM framework to add support for the MyriaNed platform (Chapter 3) and the GMAC (Chapter 2) family of protocols. The modules representing the MyriaNed platform use parameters based on the real MyriaNed hardware. For example, the simulated radio state changeover times are identical to the specifications of the nRF24L01+ radios used on the actual V3 and Chalcedony nodes.

Clocks Clearly, since we are interested in examining the synchronization behavior of GMAC, our simulated clocks are of great importance. As such, we designed OMNeT++ modules to represent the clocks found in our sensor nodes. We model each individual clock as having a constant offset from the desired clock frequency, f. That is, each node's clock will independently count ticks at a slightly faster or slower rate than f. Many real clocks behave similarly, exhibiting a relatively constant frequency given a constant ambient temperature. Hence, our simulations assume the nodes operate at constant temperature. There are more complicated and more accurate clock models that, for example, take individual temperature variations into account, but we leave such models to future research.

OMNeT++ keeps track of the global simulation time, t, while the clock module for an individual node i computes the local time, t_i. A node's local time is based on its own clock's frequency offset (f_i) and phase offset (p_i), provided as OMNeT++ simulation parameters. Thus, node i can compute $t_i = (t \times f_i) + p_i$. A node's phase

[1] `http://www.omnetpp.org`
[2] `http://mixim.sourceforge.net`

offset determines the length of time between the global start of the simulation and the start of that particular node. The frequency offset determines how much faster or slower than simulation time a node's clock runs. Unless otherwise specified, the clock at each node, i, will use a random frequency multiplier $0.99998 < f_i < 1.00002$, i.e., ±20 parts per million (ppm). This threshold was chosen to match the specifications of the real clocks used in the MyriaNed platform (Chapter 3). We call this parameter *MaxClockDrift*, because it is the differences in clock frequency offset that cause the nodes to "drift" apart and desynchronize.

Transmission power In the simulator, a node's transmission range is determined by its transmission power. By increasing a simulated node's transmission power the simulator increases the node's transmission range, and vice versa. OMNeT++ models radio transmission using a unit disc representing a node's *maximum* transmission range. The size of this disc is determined at the start of the simulation, based on the maximum transmission power parameter. During the simulation, nodes can (dynamically) set their transmission power to any value less than or equal to this maximum value. OMNeT++ uses this disc as an optimization: nodes outside this area can *never* receive the message. However, all nodes within the transmission range are potential receivers. Each potential receiver uses a sophisticated and modular analogue radio model to determine whether they can receive the message broadcast, given the sender's instantaneous transmission power and relative distance.

In all experiments presented in this thesis we set the transmission power for all nodes in the network on a per-run basis. That is, the nodes are given a maximum transmission power, and always use this as their instantaneous transmission power. We do not investigate varying transmission power during the execution of an experiment. Due to this, we can discuss the *transmission density* of a simulated run. What we mean by transmission density is the average number of nodes per transmission range, i.e., node density ($\frac{nodes}{m^2}$) × transmission area (m^2). The maximum potential transmission area ($A_{tx} = \pi r_{tx}^2$) is determined by the maximum transmission range (r_{tx}), which is in turn determined by the simulation parameter *MaxTxPower*. The node density is determined by the network topology.

4.1.2 Network topology

In order to investigate the performance of GMAC in simulation, we assess the effect of network topology on network synchronization. The network topology is determined by two primary factors. First is the *distribution* of the nodes, i.e., where the nodes are deployed throughout the simulated area. Second is the *dimension* of the simulated area. We use a rectangular area ($L \times W$) in all simulations presented in this thesis.

In this thesis, we investigate three different node distributions: static *grid* topologies, where nodes are arranged in regularly spaced rows and columns; static *random* topologies, where nodes are distributed randomly throughout the simulation area; and *mobile* topologies, where nodes are initially deployed and later move according to specified mobility patterns.

4.1.3 Evaluation

It is important to be able to quantitatively evaluate both the default GMAC protocol described in Chapter 2, as well as our suggested improvements to it. In this section we will explain what measurements we take during our network simulations, and what metrics we derive from those measurements.

Measurements

A node, i, logs a number of statistics at the beginning of each new frame, r. Two of the most important, in regards to synchronization, are the global simulation time ($t_{r,i}$) it began the frame, and its current cluster tag ($c_{r,i}$, discussed in Sec. 4.2.3). Using the logged timing data, we can see not only which nodes *are synchronized* to which other nodes (i.e., whether their active periods overlap), but how tightly they are synchronized (i.e., how much their active periods overlap). Through the recorded cluster tags, we can determine which nodes *think they are synchronized* with which other nodes. In addition to time and cluster tag, each node records its absolute position at the start of the frame, $x_{r,i}$ and $y_{r,i}$. Nodes also log a number of packet-level statistics, like the number of sent and received packets (both application and **join** packets), number of collided packets, and the number of attenuated packets (those lost due to weak/distant transmission signals). Finally, nodes will also log application data, if the application implements the logging callbacks described in Sec. 3.2.1.

Metrics

In order to evaluate how tightly synchronized the entire network is, we compute the standard deviation, σ_r, of reported start times for frame n across all nodes. By looking at how the standard deviation changes as the simulated run progresses, we can see whether the synchronization mechanism is able to reach or maintain tight temporal coupling of the nodes. Because we simulate clocks with a frequency of $32,768Hz$, one timer tick is $\frac{1s}{32768} \approx 30\mu s$, while one 28-tick slot is approximately $850\mu s$, and one 8-slot active period is about $7ms$. We consider an entire network to be *loosely* synchronized when σ_r drops below $2000\mu s$, signifying that the vast majority of nodes in the network will have overlapping active periods. If σ_r is below $300\mu s$, equivalent to a single packet transmission time, we consider the network to be *tightly* synchronized.

In order to get a better measurement of local synchronization, we can also compute the standard deviation of start times amongst the direct (1-hop) neighbors of each node, then average this deviation across all nodes. We call this the *local* σ_r, and we denote it as λ_r. This measurement is more meaningful in static scenarios, where a node will have persistent neighbors, than in the mobile scenarios presented in the next chapter, where a node's neighborhood will be constantly changing. Thus, the more mobile the nodes, the more we need to focus on tight global synchronization.

Finally, it is often useful to be able to condense an entire simulated run down to just a single number in order to easily compare the effect of different parameters on the execution. We consider one such metric to be the number of nodes that have mutually synchronized. As we study networks of different sizes, it makes sense to compute the *percentage of synchronized nodes*. For the purposes of this metric, we consider a set of nodes to be synchronized when the difference between the largest and smallest reported times for frame r is less than $\delta = 12ms$. Note that this differs from our calculation of syncgroups discussed below. In large networks, a syncgroup could span more than δ, as long as individual nodes are separated by no more than ϵ. Here we take δ to be a hard limit on the span of a group of synchronized nodes. We have chosen this particular value for δ because, particularly in a network with a large diameter, it is not necessary for all nodes to share the exact same active period (i.e., $\delta = 7ms$). Rather, it is important that nodes that are physically close are tightly synchronized, while nodes separated by larger distances can be more loosely synchronized. GMAC is designed to operate in such a globally asynchronous/locally synchronous mode, so we have chosen δ to be about one and a half active periods. For each frame r, we find the largest percentage of nodes whose reported start times fit within this window.

Groupings

Based on each node's reported time, position, and cluster tag, we can group the nodes together along these three dimensions. For example, two nodes, a and b, that report positions separated by a distance $d_{a,b}$ less than the transmission range, r_{tx}, are said to be in the same *subnet* (i.e., they are physically connected/proximal). Likewise, nodes that report start times for frame n within some ϵ of each other are in the same *syncgroup* (i.e., they are temporally connected). Finally, in cases where nodes use our optimization based on cluster tags, two nodes reporting the same cluster tag in the same frame are said to be in the same *cluster* (i.e., they are logically connected). The size and membership of each of these three types of node groupings can give us insight into what is happening in the network on a frame-by-frame basis. We use an algorithm called *DBSCAN* [50], described in Algorithm 4.1, in order to form clusters from the recorded simulation data. DBSCAN takes as input a dataset, an ϵ value which constitutes the maximum distance at which two data points can be considered neighbors, and a minimum number of ϵ-neighbors, Nbr_{min}, required to be considered part of a cluster. The DBSCAN algorithm uses this Nbr_{min} parameter in order to avoid clusters composed of long "chains" of data points $x_1, x_2 ..., x_k$ whose consecutive values differ by less than ϵ but allow for a large value of $x_k - x_i$. Requiring a minimum number of neighbors to be considered part of a cluster allows the user to vary the density of the returned clusters of data points.

We use DBSCAN to group nodes into syncgroups based on the time they started frame n. We define the absolute difference between node i and node j's reported start times for frame r as the distance metric, i.e., $dist_{i,j} = abs(t_{r,i} - t_{r,j})$. We would like to establish tight bounds and to find strongly coupled groups of nodes in order to consider them syncgroup. To that end, we determine syncgroup membership by using DBSCAN with $\epsilon = 2ms$ (about 65 clock ticks, or almost three slots) and $Nbr_{min} = 5$.

Algorithm 4.1: The DBSCAN Algorithm

```
DBSCAN(D, ε, MinNbr)
    C = 0;
    foreach Unvisited point P in dataset D do
        mark P as Visited;
        N = getNeighbors(P, ε);
        if sizeof(N) < MinNbr then
            mark P as NOISE;
        else
            C = C + 1;
            expandCluster(P, N, C, ε, MinNbr);


expandCluster(P, N, C, ε, MinNbr)
    add P to cluster C;
    foreach point P' in N do
        if P' is not Visited then
            mark P' as Visited;
            N' = getNeighbors(P', ε);
            if sizeof(N') ≥ MinNbr then
                N = N joined with N';
        if P' is not yet member of any cluster then
            add P' to cluster C;
```


These settings ensure that the temporal groupings returned by DBSCAN are composed of nodes with overlapping active periods (neighbors may differ by at most $2ms$) and are dense (i.e., not two strongly connected groups with a few nodes "bridging" them together into a single group). The number and size of syncgroups that exist in a given frame is an important measurement. The size of the largest group gives us insight into how close the network is to complete synchronization. Ideally, we would like to see our synchronization algorithms converge all nodes into a single syncgroup, regardless of their location and connectivity to other nodes. Realistically, this cannot be expected since the frame start times of isolated (groups) nodes will likely drift at a different rate than their connected counterparts. The ability to converge the network to a single syncgroup will, at least partially, depend on the number and stability of subnets and the connectivity within those subnets.

We also use DBSCAN to group nodes into physical subnets based on their reported location. We use the simple Euclidean distance, $dist_{i,j} = \sqrt{(x_{r,i} - x_{r,j})^2 + (y_{r,i} - y_{r,j})^2}$, as our distance metric, while we set $\epsilon = d_{TX}$ (the transmission range), and $Nbr_{min} = 1$. Thus, DBSCAN will return the number and size of physically connected components that exist during frame n in the network. The ideal synchronization mechanism would ensure that all nodes in a physically connected component will be synchro-

nized. That is, all nodes that could potentially communicate due to physical proximity should be synchronized so that they *can* communicate. Therefore, we consider it to be a *subnet failure* when a node that is part of a subnet S is not also in the largest syncgroup existing in S.

The *cluster* grouping tells us which groups of nodes believe they are synchronized to which other nodes. We do not need to use DBSCAN to group the data, since the logs provide us a mapping from node identifier to cluster tag. Ideally, our synchronization mechanism will ensure that all nodes that report the same cluster tag are indeed synchronized to each other. Note that this concept of a synchronized cluster of nodes exists regardless of their physical proximity/connectedness (i.e., their subnet). Similar to the subnet discussion in the previous paragraph, we consider a *cluster failure* to exist when a node in some cluster is not in the largest syncgroup that exists in that cluster. That is, each node that reports a particular cluster tag but is not actually synchronized with the other nodes reporting that cluster tag is considered a failure.

4.2 Synchronization improvements

In Section 4.4.1, we will demonstrate that the group merging behavior of GMAC is sufficient to ensure convergence for small networks, but struggles to consistently converge larger networks. The focus of this chapter is an analysis of various methods of merging large networks composed of multiple groups of nodes synchronized to non-overlapping active periods. In this section we discuss some proposed improvements to GMAC's default merging mechanisms, which were described in Section 2.4. We will analyze several distinct combinations of the improvements described below.

4.2.1 Maintenance

In this chapter we do not evaluate any new maintenance functionality, and look only at GMAC's default synchronization maintenance behavior.

4.2.2 Detection

In addition to GMAC's default method of active detection, we implemented the previously described passive detection functionality in order to allow for a comparison of the effectiveness of the two methods. In our implementation of passive detection a node listens to the whole inactive portion of its frame with probability p_l. In order to have a fair comparison between active and passive detection, we would like to spend approximately the same amount of energy in both cases. Sending a **join** message costs an amount of energy equal to the sum of the energy required to wake up the node, turn on the node's radio, broadcast a message, and turn off the node's radio

again. On the reference radio, this costs about the same as two active receive slots, so we would like to listen to two inactive slots per frame. Based on $N_{inactive} = 1162$ slots, we set $p_l = \frac{2}{1162} \approx 0.17\%$.

We also implemented a modified version of passive detection designed to augment both detection methods. This technique is based upon the notion of *superior* and *inferior* syncgroups, which is explained in the next section, 4.2.3. Normally a node will merge immediately upon discovering (either via normal active or passive detection) a superior syncgroup. However, in very large networks where many syncgroups exist before finally converging, it may prove effective to skip merging into a newly discovered "second-best" syncgroup, if there are better ones within range. In order to achieve this behavior we added a new technique called *listen before merge*. After discovering a superior syncgroup, a node executing this behavior in conjunction with active detection will listen during the entire inactive period of the frame in which the detection event occurred. With the same goal, a node using passive detection will listen during the remainder of the inactive period in which it detected a superior group. In either case, at the end of the frame, the node will merge into the *best* syncgroup it has discovered. This technique will need to significantly improve performance in order to justify the cost of an additional $T_{inactive}$ time spent with the node's radio active for each detection event.

4.2.3 Decision

Ideally, we would like the group with fewer nodes to always merge into a group with more nodes, to minimize disruption to the network. However, computing such network metrics in a decentralized fashion is a difficult problem. Even if our nodes all knew the exact size of their group, we would still need a method of breaking ties between groups of equal size. Such a tie-breaker method can also serve as the primary criterion for the group-merge decision. This may lead to suboptimal merge operations, i.e., forcing many nodes to resynchronize to match a few. Deterministic convergence, however, is more important than optimality, particularly because in a stable network the occurrence of merge operations can be assumed to be infrequent.

We propose to solve the convergence problem using a relation, $\succ_c$, based on *cluster tags*. A cluster tag is simply an identifier used by a group of nodes that share a common active period, i.e., a syncgroup. Nodes that have the same cluster tag are said to be in the same *cluster*. Note, however, that two nodes in the same syncgroup need not be in the same cluster, e.g., because the best cluster tag has not propagated throughout the entire syncgroup yet. Similarly, two nodes in the same cluster need not be in the same syncgroup, e.g., because they were synchronized but have drifted apart.

In our solution, a cluster tag C is composed of two integers, an *ID* and an *epoch*, and written $C = \{C.id, C.epoch\}$. We assume that all nodes have a unique identifier and, upon starting, nodes initially use their own unique identifier and an epoch of 0 for their cluster tag. We will defer discussion of the use and function of epochs to the next chapter. Nodes in the `INITIAL_LISTEN`, `SAY_HELLO`, or `KEEP_LISTENING`

states (described in Section 2.4.1) do not respect the ordering of cluster tags, and will respond to any received message by synchronizing with its sender. Once in the state `SYNCHRONIZED`, a node will strictly respect the ordering of cluster tags by following the protocol described here.

As discussed earlier in this section, nodes use a relation, $\succ$, to determine whether they should join a newly discovered syncgroup. Here we define a new relation, $\succ_c$, that allows a node to make a deterministic decision. A node can compare its own cluster tag, A, with a received tag, B, using this relation: $B \succ_c A$ if $B.id > A.id$ (again, ignoring epochs for now). A synchronized node will always adopt a superior cluster tag received from other nodes in the *same* syncgroup. That is, if a node with cluster tag A receives an **application** message with cluster tag B during its active period (signifying that the sender's active period overlaps with its own) and $B \succ_c A$, it should discard its old tag and adopt the tag B. Similarly, if this node in A detects a node with cluster tag C from a *different* syncgroup (either by hearing a **join** message during its active period, or by overhearing any message while listening during its inactive period), it can simply compare its own cluster tag to that of the other node. If $A \succ_c C$, its own ID is higher and it is already in the superior cluster and can ignore the message. However, if $C \succ_c A$, the other group's cluster ID is higher and the node can deterministically decide that it should merge its inferior cluster into the other and react accordingly. By assuring that the nodes in a syncgroup with a superior cluster tag never merge into a syncgroup with an inferior cluster tag, we can eliminate the cycling problem in GMAC's decision mechanism. The relation $\succ_c$ provides for all three properties, including the transitivity missing from GMAC's default relation, $\succ_t$. Without cycles, all other syncgroups should eventually merge into the group with the best cluster tag.

4.2.4 Notification

In order to add notification functionality to GMAC, we have added a *merge* field to the header of **application** messages. This allows a node to notify its neighbors when it detects a superior group. After discovering a group with a better cluster tag, a node can record the time difference, or offset, between its own group and the one with the superior cluster tag. Then, rather than immediately merging into the new group, it can stay synchronized to its current group for one more frame, in order to communicate with its neighbors and inform them about the new superior group. By sending this merge offset along with its message in the following frame, its current neighbors can be made aware of both the existence and the offset of this superior group *without* the need to detect it on their own. This notification should greatly reduce the time and energy spent on detection, particularly at low duty cycles which reduce the probability of detecting other clusters.

4.3 Experimental setup

We conduct our simulations using the the OMNET++ simulation environment, as described in Section 4.1. Here we will briefly describe the specific setup and parameters

Table 4.1: Static network topologies investigated in this chapter

N	Nodes (N^2)	Dimensions	Distribution
4	16	$320m \times 320m$	80m Grid & Random
8	64	$640m \times 640m$	80m Grid & Random
16	256	$1280m \times 1280m$	80m Grid & Random
32	1024	$2560m \times 2560m$	80m Grid & Random
64	4096	$5120m \times 5120m$	80m Grid

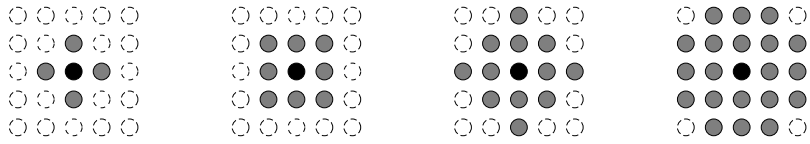
used for the experiments presented in the chapter. As we are primarily interested in GMAC's network synchronization behavior, we focus on the following simulator parameters: clock drift, network topology, and transmission density. In addition, we investigate a number of GMAC *configurations*, or modes of operation. Finally, we look at several different simulation *scenarios*, or general activity patterns the nodes will follow.

4.3.1 Simulator parameters

Clocks In this chapter, we are interested in investigating GMAC's synchronization maintenance, so we also look at a variety of settings for *MaxClockDrift*. We simulate clocks with accuracies from ±1 ppm to ±100 ppm, as well as a number of experiments with the default setting of ±20 ppm.

Network topology In this chapter, we investigate the effect of topology on network synchronization. In all of our experiments in this chapter the nodes are distributed in one of two patterns: an $N \times N$ *grid* or uniformly at random. In the grid topologies, N^2 nodes are deployed in N rows of N columns placed $80m$ apart (see Table 4.1). For some experiments it is very important that the networks we examine are connected, because otherwise complete synchronization would be impossible. Though not the most representative of real-world deployments, grid topologies allow us to directly observe the effects of various transmission densities with a uniformly connected network topology. In the random topologies, N^2 nodes are deployed at random locations within an area of the same dimensions as an equivalent grid topology (i.e., with the same number of nodes). Random topologies present more difficulty in synchronization as there will generally be some very highly connected areas and some less connected areas, and potentially even physically isolated nodes. These random topologies, however, are more representative of the type of topologies that will be encountered in the real world, and hence of more interest to our investigation.

Transmission density We have chosen the transmit power values of $10mW$, $20mW$, $40mW$ and $80mW$ based on our grid spacing of 80 meters, and unless otherwise specified, use $20mW$ as our default setting. In Fig. 4.1 we show a group of nodes spaced



(a) 10mW (b) 20mW (c) 40mW (d) 80mW

Figure 4.1: Graphical representation of the four simulated transmit ranges for nodes arranged in an 80m grid

80m apart, depicting the four transmit power level ranges from the perspective of the sender (black) and the potential receivers (gray). This parameter strongly influences the connectedness of a given topology.

4.3.2 GMAC configurations

In order to facilitate discussion of GMAC's behavior with various improvements (discussed in Section 4.2) switched on or off, we will analyze several specific combinations, called *configurations*.

- <**Active**> This is the default GMAC behavior, as described in Section 2.4.
- <**Active+Ids**> The same as <*Active*>, but using cluster ids (Sec. 4.2.3) in order to make consistent merge decisions.
- <**Passive+Ids**> Purely passive detection with $p_l = 0.17\%$, using cluster ids. As explained in Section 4.2.2, the value 0.17% is chosen to match the power usage of active detection.
- <**Active+Ids+MergeMsgs**> The same as <*Active+Ids*>, but nodes do not immediately join newly discovered clusters, rather they wait one frame in order to send merge messages (Section 4.2.4).
- <**Active+Ids+Listen**> The same as <*Active+Ids*>, but nodes do not immediately join newly discovered clusters, rather they listen for a whole frame in order to discover the best cluster in range (which we also refer to as *listen-before-merge*, Section 4.2.4).
- <**Active+Ids+MergeMsgs+Listen**> This configuration uses active detection, cluster ids, and uses both listen-before-merge and merge messages.

4.3.3 Scenarios

We utilize four different *scenarios* in order to evaluate different aspects of the synchronization behavior:

Synchronous start The synchronous start scenario is the most straight-forward that we investigate here. All nodes start up at the exact same time, $T_{start} = 0.0s$ in the SYNCHRONIZED state. In this state, nodes act as though they are already synchronized with their neighbors, and immediately begin performing the GMAC protocol and exchanging application messages. Nodes will execute the GMAC duty cycle of eight active slots followed by a long inactive period, dependent upon the length of the frame.

Asynchronous start In the asynchronous start scenario, ll nodes start up at a random time $1s \leq T_{start} \leq 15s$ in the INITIAL_LISTEN state. In this state, nodes know they are unsynchronized and search for a synchronized group of nodes to join. Initially they will continuously listen for a message for a random initial period T_{catch}, lasting between one and two frames. If they do not "catch" (by hearing a message) before the end of this extended frame, they then broadcast a single **HELLO** message. After sending their message, they switch back to continuous listening mode and remain in that mode. When a INITIAL_LISTEN node hears a message, it will enter the CAUGHT state, and try to synchronize its next frame with the node (and cluster) that it heard. Here, nodes that use the *listen-before-merge* option will continue listening until the end of their frame, rather than going to sleep. Once the node has performed its frame-length adjustment, it will enter the SYNCHRONIZED state, and will assume that it is synchronized. When in a normal, synchronized state, nodes will execute the GMAC duty cycle of eight active slots followed by the long inactive period. Note that nodes that use cluster ids will ignore them while INITIAL_LISTEN, because otherwise a node with a high id may stay isolated and silent, ignoring all its neighbors that happen to have lower ids. Once a singleton node has found an initial cluster and synchronized with it, it will respect the ordering of cluster ids from then on.

Singleton We simulate two different variations of a *singleton* scenario: a singleton cluster with an *inferior* id detecting and merging into the established superior cluster, and an existing cluster detecting and merging into a singleton cluster with a *superior* id. We call these scenarios *SingletonWorst* and *SingletonBest*, respectively. These are by far the simplest set of merge scenarios that we investigate. That is particularly so in the *SingletonWorst* case, where the desynchronized node has an inferior cluster id. In this case, the network will converge after that single node detects the other cluster, and merges into it. In the *best* case, the single desynchronized node has a superior id, and must get all other nodes to merge into its cluster. In both variations of this scenario, the isolated node is located in the top-left corner of the grid. We have chosen this location because it maximizes the distance (hops) that the synchronization information must travel to reach all nodes in the network.

Cluster merge The cluster merge scenario is designed to give us insight into the behavior of cluster merging in the case of multiple synchronized clusters. Here we only look at the 32×32 (1024-node) topology. The sixteen columns of nodes on the right-hand side of the grid begin as one synchronized cluster, and the fourteen columns of

nodes on the left-hand side begin as another. For the first five seconds of the simulation, the rightmost two columns of the left-half of the grid are inactive. At $T_{sim} = 5s$, these 64 nodes start up as a third synchronized cluster. The left cluster has id 1, the middle cluster has id 2, and the right cluster has id 3. This final scenario is more complex than the singleton one, but more straight-forward than the asynchronous start.

4.4 Simulation results

In this section, we present the results of our experiments simulating the GMAC layer in the OMNET++ environment.

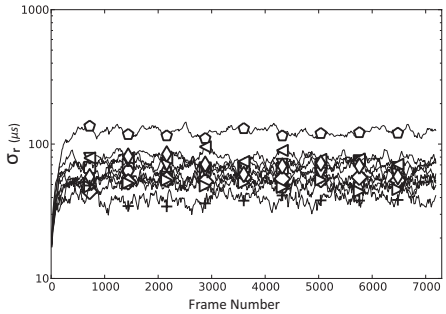
4.4.1 Maintenance

For this series of experiments, our primary metric for measuring the level of synchronization in the network is the standard deviation of the node start times. This is the σ_r metric described in Section 4.1.3. Note that the best synchronization level we can realistically hope for is $\frac{t}{2} \simeq 15\mu s$, or half the smallest adjustment that the GMAC can make in its effort to compensate for the drifting clocks. Consistent and stable synchronization to the level of a single clock-tick is quite good, though, and indicates that we could reduce T_{guard} significantly below its current value of $300\mu s$ in order to save energy.

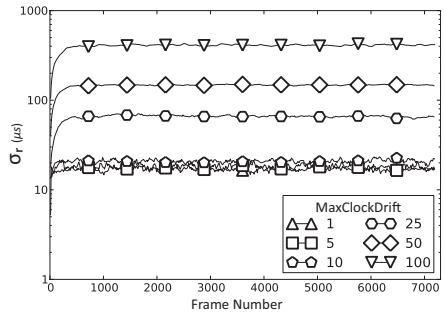
We begin by looking at Fig. 4.2a, depicting a set of individual simulated runs. Each line is the result of a single run on a 256-node grid topology, with the standard deviation plotted per frame. These simulations used the default transmission power of $20mW$ and *MaxClockDrift*=$25ppm$. Here we can see much similarity between individual runs. The median algorithm maintains a steady-state where it compensates for the clock drift between the nodes.

In Fig. 4.2b, each line shows the average standard deviation for each simulated frame across the 10 experimental runs for each *MaxClockDrift* setting. We explore the effects of different clock drifts on the median algorithm, using the same topology and transmit power settings as the previous set of results. Thus, the 10 lines from Fig. 4.2a comprise the single line in Fig. 4.2b at $25ppm$. Here we can see how the median algorithm copes with various clock settings. More clock drift (higher values of *MaxClockDrift*) between nodes leads to progressively looser synchronization. That is, as clocks drift apart faster and faster, the median cannot maintain the same level of synchronization between nodes. This can be seen by the increased standard deviation at higher drift settings.

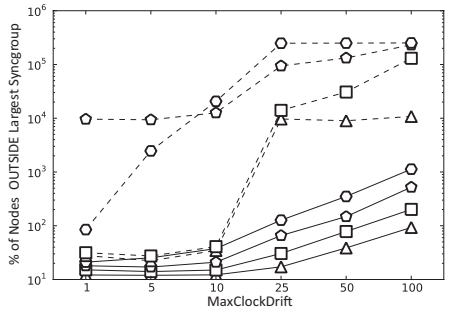
Finally, in Fig. 4.2c we see the aggregation of a large amount of experimental data. Each data point represents the average standard deviation for the last half of the 10



(a) 10 individual runs of same settings
Topology: 256-node Grid, *MaxClockDrift*: 25*ppm*



(b) Average over 10 runs for each *MaxClockDrift*
Topology: 256-node Grid.



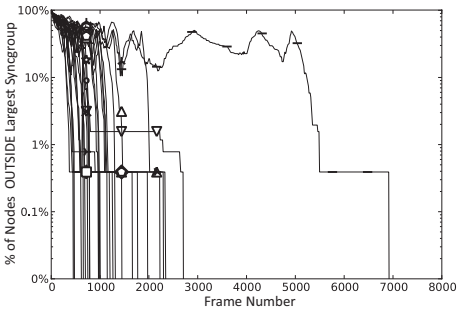
(c) Converged offset standard deviations for combinations of *MaxClockDrift* and *Topology*.
1024 (hexagon), 256 (pentagon), 64 (square) & 16 (triangle) nodes; Random (dashed) & Grid (solid) topologies.

Figure 4.2: Variation in frame start times, synchronous start

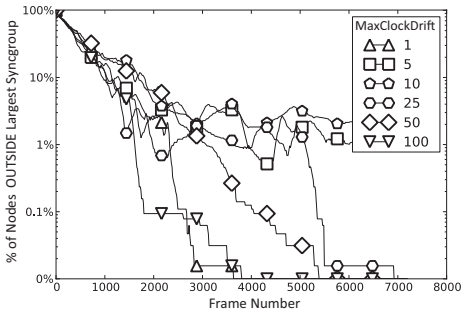
runs of a particular parameter setting. We plot the clock drift settings along the *x*-axis and average converged standard deviation for the 10 runs of each *MaxClockDrift* value. Thus, the single points along the line {*Grid*, *256*} show the average value over the last 30 minutes of the 10 runs represented by the individual lines from Fig. 4.2b. The solid lines show *grid* topologies, while the dashed lines show *random* topologies and, as expected, the random topologies show much more variability.

4.4.2 Merging

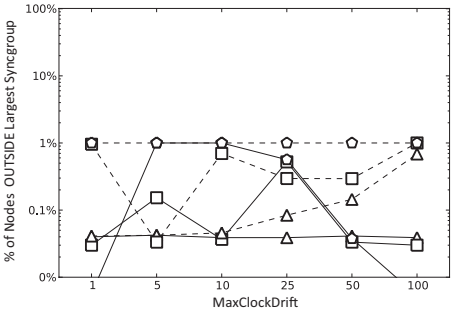
We look at three different scenarios in order to evaluate the three different aspects of merge behavior. First, to examine the *decision* aspect of merging, we use *AsynchronousStart*. In the *Singleton* scenarios, we explore the *detection* aspect of group merging. Finally, using the *ClusterMerge* scenario, we look more generally at two established clusters merging. For this scenario, we can investigate how the size, topology and distribution of the clusters affects the merge behavior, as well as examine the effects of our *notification* improvements.



(a) 10 individual runs of same settings
Topology: 256-node Grid, *MaxClockDrift*: 25*ppm*.



(b) Average over 10 runs for each *MaxClockDrift*
Topology: 256-node Grid.



(c) Per-Parameter Per-Topology Average. Note that we use the same lines and markers as Fig. 4.2c

Figure 4.3: Percentage of unsynchronized nodes, asynchronous start

Default GMAC merge behavior

Here, our metric for analysis is the fraction of nodes *not* belonging to the largest synchronized group. If the percentage of *synchronized nodes* (as discussed in Sec. 4.1.3) is denoted P_{sync}, then our metric here is $P_{unsync} = 100\% - P_{sync}$. Not that we choose to plot the results in this manner so it is clear to the reader when 100% of the network has converged. By plotting the percentage of *unsynchronized* nodes, the lines clearly go to zero. However, plotting the percentage of *synchronized* nodes, it is sometimes unclear whether all or only 99% of the nodes have converged. At each frame, we calculate the percentage of all simulated nodes that are *outside* (i.e., not synchronized with) the largest syncgroup. As such, we should expect this metric to begin at or near 100% and fall towards 0%. In Fig. 4.3, we show the results of our asynchronous start experiments. In general, most runs behave as expected and rapidly converge to a single network. However, in some runs, we find that separate syncgroups can exist for a long time, to the point where they never form a single network during the entire simulated hour. We can see just such behavior in Fig. 4.3a, which depicts the results of a set of twenty-five 256-node grid runs with a maximum clock drift of ±25*ppm*. In Fig. 4.3b, we show the average of 25 such runs for each *MaxClockDrift* setting. As

previously explained, the behavior of the join mechanism is probabilistic, and this can be clearly seen in the variability of the simulated results.

Finally, in Fig. 4.3c we plot (analogously to Fig. 4.2c) the average fraction of nodes outside the largest syncgroup over 25 runs for each setting, considering only the last half of the frames of each run. The settings correspond to all combinations of clock drifts and topologies we experimented with. Clearly, in all of our simulations GMAC eventually achieved synchronization of at least 99% of the participating nodes, irrespectively of the topology and maximum clock drift.

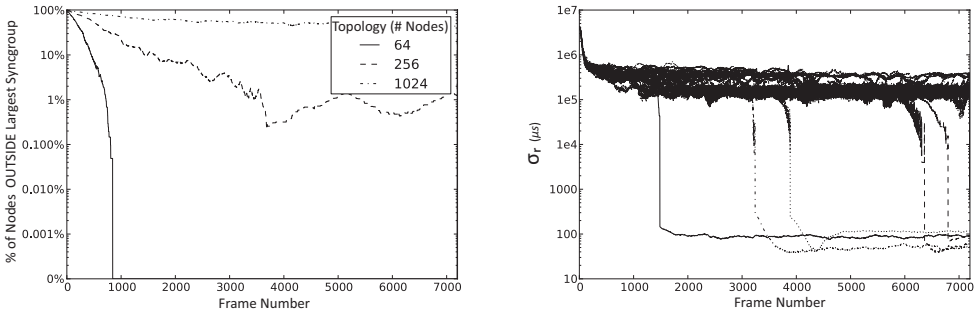
Decision

We begin with the *AsynchronousStart* scenario for two reasons. First, this is the scenario that initially pointed at failings in GMAC's synchronization mechanisms, so it makes sense to reproduce those results here. Second, by demonstrating the performance of all our test configurations in the most demanding circumstances, we can focus our analysis on the best few.

In Figure 4.4 we see the performance of the <*Active*> GMAC configuration. Figure 4.4a shows a plot of the standard deviation of start times as a function of the frame number, averaged across 32 runs. We see that GMAC's <*Active*> configuration works acceptably in the small 64-node topology, converging all nodes to a single syncgroup in an average of about 1000 frames (about 8 minutes). However, it does not consistently synchronize the 256-node and 1024-node test cases. In the 256-node networks, the majority of all nodes synchronize to the largest group, but not all runs synchronize. In Figure 4.4b, we see a composite plot showing that only six of the thirty-two 1024-node runs converged to a completely synchronized network, and in those cases it often took almost a full simulated hour.

To examine the cause of the <*Active*> configuration's difficulties, we look at the results of some individual runs that clearly show the problem. The three graphs on the left side of Figure 4.5 show the difference in μs between each node's start time and the average start time for the frame on the x-axis. Those on the right show the percentage of nodes *outside* the largest group, calculated as described earlier. Figure 4.5a shows a 256-node run that properly converges to a single syncgroup around frame 2500. The individual syncgroups can be seen previous to that time as clusters of points, converging to bounds of about $\pm 100 \mu s$. Figure 4.5b is another view of the same data, showing the nodes outside the largest syncgroup dropping to zero. The results of 1024-node runs show similar initial behavior, but generally fail to converge. For an example of a non-converging run, see Figures 4.5c and 4.5d. These results demonstrate the problem of cyclic ordering of syncgroups described in Section 2.4.2. The <*Active*> merge protocol can only deterministically synchronize two groups. If there are more than two groups a cycle may exist, preventing convergence for an arbitrarily long time. The severity of the problem is dependent on the size of the network, and will make it unsuitable for very large networks.

In order to solve this *decision* problem, we have proposed using cluster ids, described in Section 4.2.3. In Figures 4.5e and 4.5f, we show the performance of the



(a) Performance of <*Active*> for increasing network sizes

(b) Per-frame standard deviation of start times for 32 runs of <*Active*>, 1024-node grid

Figure 4.4: Group merging using the *active* detection

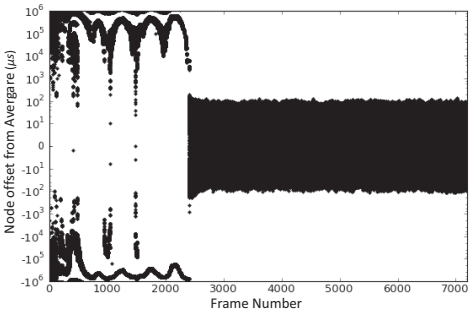
<*Active+Ids*> configuration for a representative 1024-node run. The general merge mechanism is the same as <*Active*>, but nodes use the cluster ids to reliably decide whether to join a discovered group, rather than arbitrary timing heuristics. When compared to the results for the <*Active*> configuration, the ids clearly provide for superior performance. In Figure 4.6a we show the average performance of the <*Active+Ids*> configuration, and we can see that it is able to consistently synchronize a 1024-node network in a little over 1000 frames. This is the same number of frames it takes the <*Active*> configuration to synchronize a 64-node network. Based on these findings we can eliminate the <*Active*> configuration from further study.

Detection

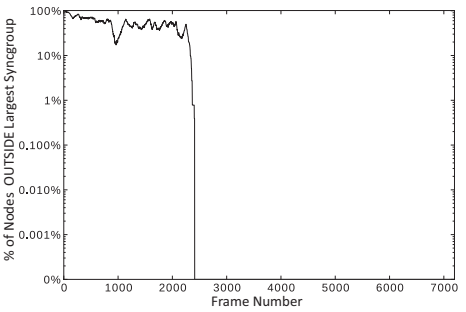
GMAC currently uses an active detection mechanism, as described earlier. Here we will examine the performance of the new passive detection configurations we have designed, i.e. <*Passive+Ids*>, in order to see whether they are superior to the existing active mechanism. We begin with more results from the *AsynchronousStart* scenario, then proceed to evaluate the *SingletonBest* and *SingletonWorst* scenarios.

In Figure 4.6c we show the average behavior of <*Passive+Ids*> over 32 simulated runs. The performance is clearly superior to that of <*Active*> (Fig. 4.4a) on larger topologies, though it does take longer to converge for small networks, as seen from the 64-node topology results. At larger network sizes, the consistent ordering of clusters turns out to be an essential element that overcomes passive detection's inherent performance disadvantage. However, when compared to the <*Active+Ids*> configuration (Fig. 4.6a), there is really no competition. Active detection is several times faster than passive detection, because of active detection's ability to recruit multiple inferior nodes with a single broadcast.

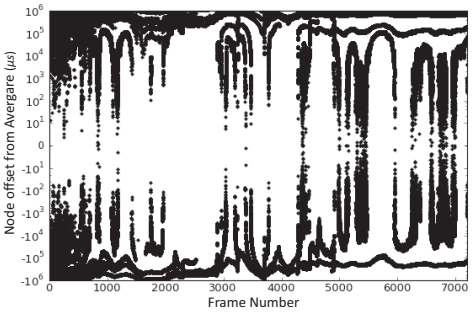
We evaluate our other proposed improvement to detection, *listen-before-merge*, using the same chaotic start scenario in Figure 4.6b. Though hard to see in the graphs,



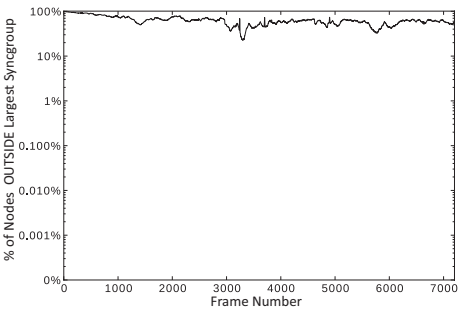
(a) A 256-node run, showing proper synchronization



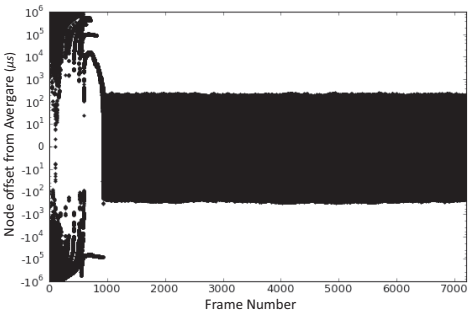
(b) The same simulated run as Fig. 4.5a, showing nodes outside of the largest syncgroup



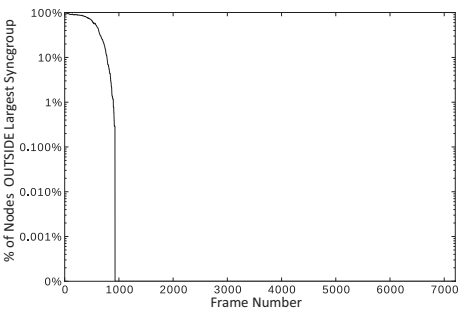
(c) Example 1024-node simulation that fails to reach complete synchrony



(d) Nodes outside the largest syncgroup in the same 1024-node run as Fig. 4.5c

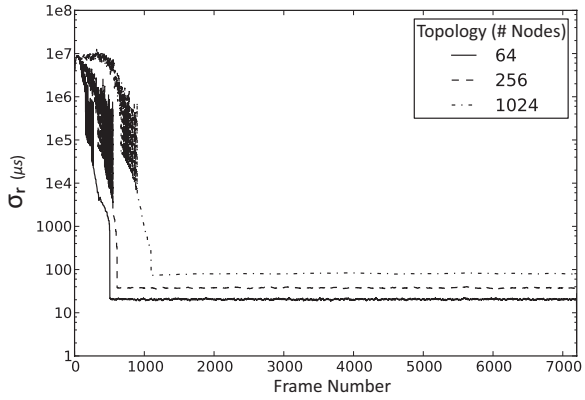


(e) A 1024-node <*Active+Ids*> run demonstrating correct synchronization

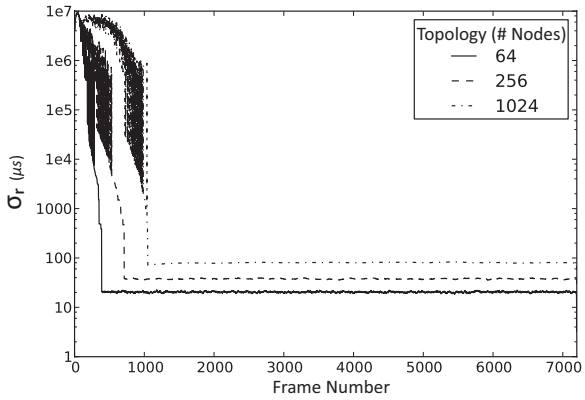


(f) Nodes outside the largest Syncgroup in the same 1024-node run as Fig. 4.5e

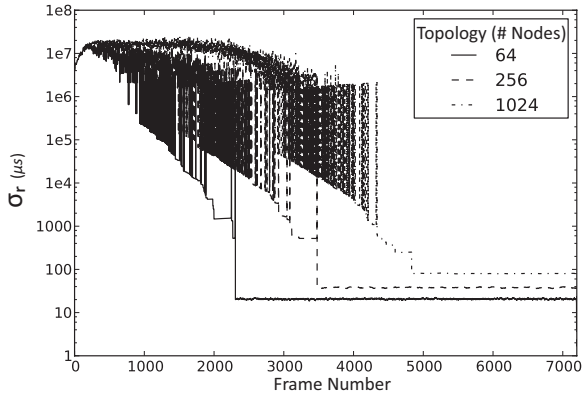
Figure 4.5: The problem with <*Active*>'s merge mechanism and a proposed solution, <*Active+Ids*>



(a) <*Active+Ids*>



(b) <*Active+Ids+Listen*>



(c) <*Passive+Ids*>

Figure 4.6: Comparison of configurations using *cluster IDs*

the *listen-before-merge* behavior does give a performance improvement on the largest test-case, but it is quite minimal. This scenario should be the one in which this behavior provides the most benefit, as there may be many different groups in a node's transmit range. However, this additional listening can only help if there are more than one *superior cluster* in range, and even then, it will only help if the node detects a second-best group to begin with.

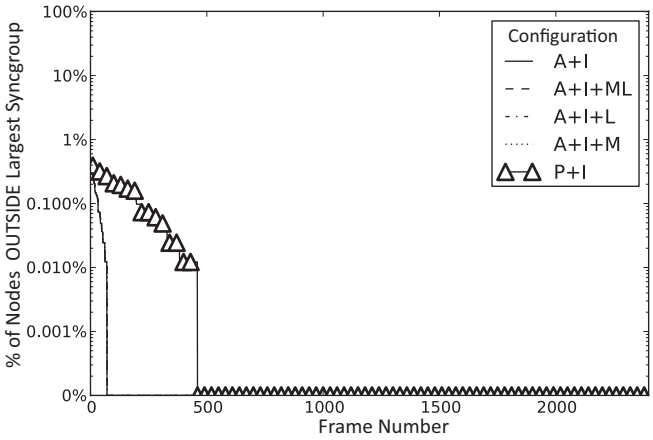
We continue our evaluation by looking at our singleton cluster scenarios. In Figure 4.7 we can compare the performance of <*Passive+Ids*>, labeled 'P+I', to that of two of our active configurations: <*Active+Ids*> and <*Active+Ids+Listen*> (labeled 'A+I' and 'A+I+L', respectively). Please note that the lines labeled 'A+I+M' and 'A+I+ML' are not discussed until Section 4.4.2. We simulate these configurations using both versions of the singleton merge scenario and a 1024-node topology. Each graph shows the percentage of nodes that are not synchronized to the largest cluster as a function of the simulated frame. First, in Figure 4.7a, we examine <*SingletonWorst*>, where the single node has an inferior cluster id and should therefore join the other cluster (containing all other nodes). All active detection configurations handle this simple scenario identically, and manage to synchronize the isolated node in an average of about 100 frames. Passive detection, on the other hand, requires five times as long to synchronize.

Finally, we look at the <*SingletonBest*> scenario. In this scenario, the isolated node has a superior cluster id, so all other nodes must leave their cluster and join the singleton cluster. In Figure 4.7b, we see that all the configurations using active detection eventually synchronize the network, but passive detection again performs poorly and consistently fails to synchronize the network. Unsurprisingly, the *listen-before-merge* optimization has not helped in this scenario, as there are no other clusters for it to detect. In fact, it performs slightly *worse* than the simple <*Active+Ids*> configuration. The additional listening often causes delays in synchronization as nodes listen for better groups that do not exist. Additionally, the *listen-before-merge* behavior costs a significant amount of energy. Using a duty cycle of 1%, a full frame of listening costs about the same energy as 100 'normal' frames. Given the high cost and low success rate of *listen-before-merge*, we can discontinue investigating the <*Active+Ids+Listen*> configuration.

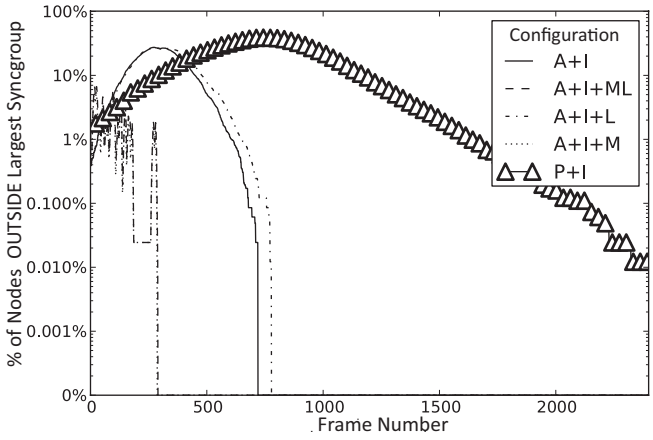
Notification

In this part, we delve into our final performance enhancement: notification. When one node detects another cluster and decides to join it, notifying his already synchronized neighbors can save a lot of energy. Each neighbor alerted by a merge message can be spared many frames of operating in a dying subset, with few or no neighbors.

We begin by referring again to Figure 4.7b. The reduction in the time required to reach complete synchrony for the <*Active+Ids+MergeMessages*> and <*Active+Ids+MergeMsgs+Listen*>, labeled 'A+I+M' and 'A+I+ML', is clearly significant. These two configurations perform almost identically and can synchronize all nodes in the simulated 32×32 grid almost five times faster than the configurations without merge messages. The best



(a) SingletonWorst



(b) SingletonBest

Figure 4.7: *Passive* detection compared to *active* detection using our two Singleton scenarios

part about the merge messages is that they are essentially free. The only cost is the overhead of a two-byte merge offset with each application message.

In order to further evaluate the performance of the merge message optimization, we turn to our final scenario, *ClusterMerge*. In this scenario we simulate the merging of three separate clusters in a 1024-node grid. Figure 4.8 shows the results of our three remaining configurations. The basic <*Active+Ids*> configuration performs quite well, taking only 500 frames on average to converge to a single cluster. However, the configurations with merge messages perform even better and reach synchrony in less than 100 frames.

We evaluate the effects of density and duty-cycle on synchronization, by returning again to our original scenario of an asynchronous network start. In Fig. 4.9a we examine the role of density on group merging by simulating each of the four

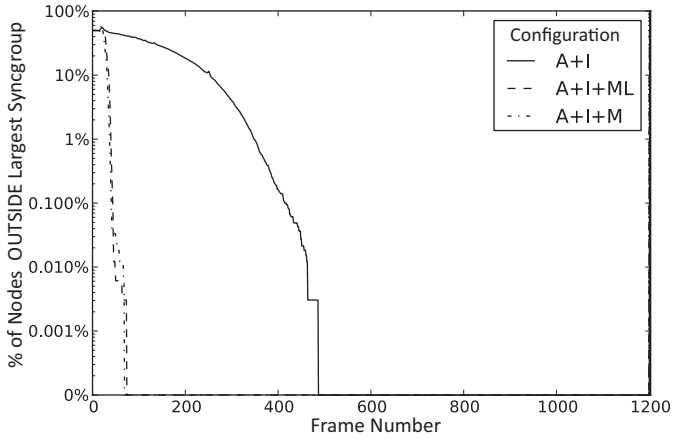
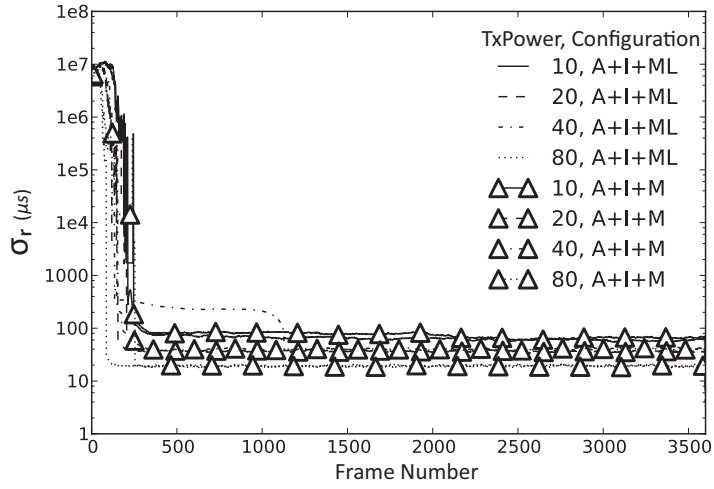


Figure 4.8: Merging three separate clusters in the *ClusterMerge* scenario

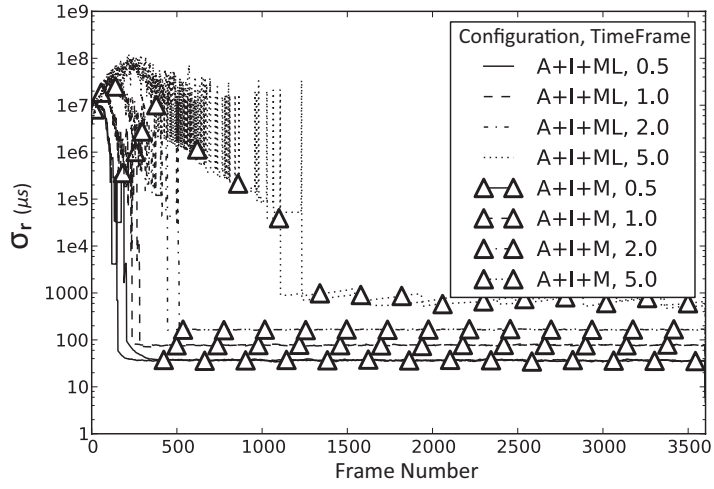
transmit power settings described in Section 4.3. The density of the network is determined by the nodes' transmit power, so we can directly investigate the effects of network density on synchronization. Here we evaluate only the two most promising configurations: <*Active+Ids+MergeMsgs*> and <*Active+Ids+MergeMsgs+Listen*>. The question we would like to answer is: does *listen-before-merge* offer any significant benefit in this scenario? The potentially large number of syncgroups present in the asynchronous start-up scenario provides the best opportunity for this modification to demonstrate its value. Unfortunately, at least in the case of varying network density, there is no clear advantage to performing a long listen before merging. Both configurations perform quite similarly, and as mentioned previously, the cost of performing these listening periods is prohibitively high without strong evidence of better performance. From the results of these simulations, it seems that network density has only a minor effect on the performance of the synchronization mechanism.

The results of our set of experiments extending GMAC's frame time are shown in Figure 4.9b, and show how the nodes' duty-cycle affects synchronization. Our default frame time to this point has been $\frac{1}{2}s$, giving a duty cycle of about 1.5%. We now simulate longer frame times with the same 8 slot active period yielding progressively lower duty cycles, with the lowest being about 0.15% at a frame length of 5*s*. Lowering the duty cycle reduces the probability of detection, as discussed earlier, and that effect is clearly visible in our results. As the total frame time increases, the effects of clock drift are magnified as well, since nodes have less frequent opportunities to synchronize their clock with those of their neighbors. This behavior is clearly evident in our results, particularly at the highest frame length setting of 5*s*. Using such a low duty cycle drastically lowers the probability of detecting other synchronized groups, though GMAC copes with this using frame lengths up to 2*s*. Furthermore, we again find no evidence that the *listen-before-merge* optimization is providing any noticeable performance benefit on top of that provided by the merge messages alone.

Finally, we have one last set of simulations using an even larger topology to test the scalability of our best configuration, <*Active+Ids+MergeMsgs*>, as well as



(a) Experiments using 4 different transmit power settings
(higher is more dense)



(b) Experiments using 4 different frame length settings
(higher is lower duty-cycle)

Figure 4.9: The effects of density and frame length using the *AsynchronousStart* scenario

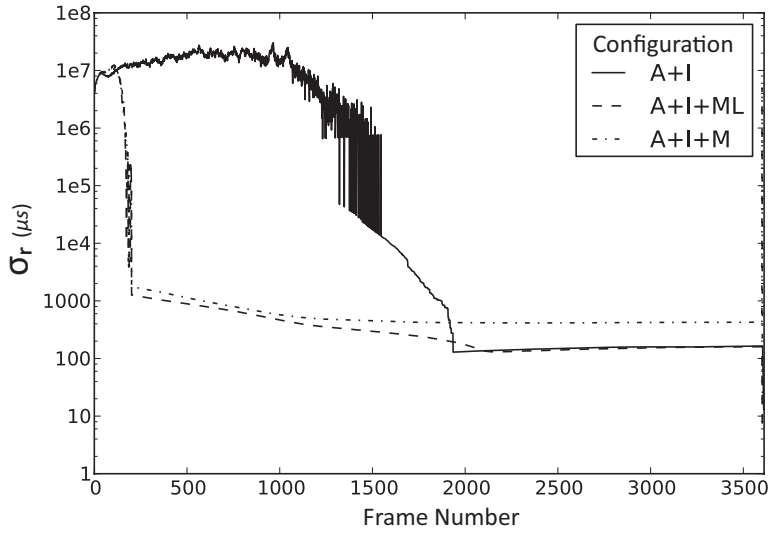


Figure 4.10: Our largest topology, a 64×64 grid of 4096 nodes

<*Active+Ids*> and <*Active+Ids+MergeMsgs+Listen*>. As shown in Figure 4.10, all these configurations can adequately handle the simulated 64×64 node grid. The performance advantage granted by the merge messages is again evident. Both configurations that include merge messages synchronize the entire network in an average of about 250 frames, while the configuration without these messages takes approximately 2000.

4

4.5 Conclusions

The main contribution of this chapter is a detailed evaluation of GMAC's method of synchronization maintenance and several proposed methods of merging separately synchronized groups of nodes. The results of our simulations show that the problem *is* solvable, and our solution can be used to achieve remarkably low duty-cycles, even with relatively inaccurate clocks. Our simulations have shown that GMAC is capable of synchronizing all nodes in a network so that they share a common active period, and doing so in a decentralized manner.

Furthermore, simulations not presented indicate that duty-cycles as low as $\frac{7ms}{5s} = 0.14\%$ are possible using these mechanisms. These simulations are not presented in this chapter because the results, while good, are not particularly interesting. Increasing the frame length (and thus decreasing the duty cycle) does not appreciably affect the network synchronization. We simulated frame lengths up to five seconds, but found no changes in the synchronization behavior. Certainly there is an upper bound on the frame length, mostly determined by the relative offset between the

nodes' clocks. Increasing the frame length does affect the behavior of passive detection, since the detection probability is determined by the duty cycle used. The results shown in this chapter represent a *best case* scenario for passive detection, with a comparatively high duty cycle of 0.7%.

The results presented here indicate that the simple median synchronization mechanism provided by GMAC is capable of adaptively maintaining good clock synchronization, even at very large network sizes. Simulations show that the median algorithm can compensate for clock frequency offsets commensurate with what can expected from real clock components. Furthermore, simulations show the median algorithm can maintain synchronization in networks as large as 4096 nodes, and possibly beyond. This level of scalability is an absolute necessity in the setting of large social ad-hoc networks. It also seems that for smaller networks, GMAC keeps nodes tightly coupled. This indicates that there may be much energy to be saved by reducing the transmission guard time.

All configurations using cluster IDs eventually synchronize the entire network, with the only difference being the time each configuration takes to do so. We demonstrated that while passive detection does consistently converge the network, it can take far longer than using active detection. Additionally, we demonstrated that the combination of active and passive detection can offer small performance benefit, but will generally not outweigh the additional energy cost. We have further shown that a simple notification message can drastically reduce the time for a network to reach a synchronized state, by as much as a factor of eight on our 4096-node topology. These two small modifications to GMAC's current behavior radically increase its suitability for large scale networks. The key insight is that as synchronized groups build up, merge messages allow GMAC to leverage the inferior cluster's existing synchronization to rapidly merge *whole clusters*, not just individual nodes. Combined with a total ordering of clusters to solve the problem of which group to join, large and complicated networks can be synchronized in just a few minutes.

4

5. Mobile networks

In this chapter, we will continue the thread from the previous chapter. We again evaluate several aspects of GMAC's network synchronization through the use of the simulation environment described in Section 4.1. The main difference from the previous chapter's evaluation is that we now look at GMAC's performance in mobile network topologies.

In addition to the inclusion of node mobility, we also look at several new potential synchronization improvements, based on our evaluation of GMAC's behavior and performance in the previous chapter. Something we learned from our initial investigations was that the synchronization maintenance aspect generally works well enough, and in a wide variety of different network conditions. One of the important insights from the previous chapter is that with very low duty cycles, the probability of one group detecting a different, separately synchronized group of nodes is proportionally low as well. Thus, we should take advantage of messages that are "lucky" enough to travel from one syncgroup to another. Hence, this chapter will focus mainly on the group merging aspect of network synchronization, as discussed in Section 2.4.

In this chapter we will evaluate the maintenance and merging behavior of GMAC in a wide variety of settings. We look at both static and mobile topologies, generated to resemble several realistic mobility patterns. We also investigate a range of different density settings on these same topologies. By adjusting the nodes' transmission power, we can affect the overall connectivity of the network. Additionally, as the sensor nodes use broadcast-based communication, they need to be careful not to overload the medium. Too many nodes broadcasting using too few transmission slots means many messages will share the same slot and the number of collisions can be high. This will decrease the utility of the network and the reliability of message exchange. It is important to evaluate GMAC's behavior in both high and low density scenarios.

The contents of the rest of the chapter are as follows: We begin by proposing a number of potential improvements to the synchronization mechanisms provided by GMAC in Section 5.1. We then explain the environment, measurements and metrics we use to evaluate our algorithms in Section 5.2. Following that, we present our experimental results and evaluation in Section 5.3, before finally concluding in Section 5.4.

5.1 Synchronization improvements

In this chapter, we will evaluate GMAC's synchronization in the face of various mobile network topologies. After encountering initial problems with GMAC's default synchronization behavior (described in Sec. 2.4), we came up with a number of potential improvements in the previous chapter. In addition to analyzing the performance of those synchronization behaviors in the context of node mobility, we add several new behaviors which we will describe in this section.

5.1.1 Maintenance

Though GMAC's median algorithm maintains synchronization well when there are neighbors for a node to communicate with, it will obviously fail to make any corrections in the absence of neighbors. In this situation, GMAC will fall back to simply letting each frame's duration be determined solely by the rate of its local clock. If the node is isolated (i.e., without neighbors) for only a few frames, this will not present a problem. However, if the node remains isolated for a long period of time, the inherent difference in clock frequency will cause the node's active period to become unaligned with those of the other nodes in its syncgroup.

For this reason, we have introduced a *history* mechanism in which a GMAC node can record its past history of frame length adjustments. In frames where a node adjusts the length of its frame in response to messages from at least two neighbors, the node can record this adjustment in order to develop an approximation of its own clock's offset from an ideal, or *true*, clock. Over time, this approximation should approach the actual clock frequency offset experienced by that node. We choose to discard adjustments based on the timing of a single neighbor in order to avoid a dependency on a particular neighbor node. In a subsequent frame where the node hears no messages, it can adjust its local frame length based on the computed approximation of its local drift, rather than not making any adjustment at all. Using this mechanism, we anticipate that GMAC nodes will be able to remain aligned with their syncgroup, even when out of contact with other group members for long periods of time. The nodes implement this behavior by computing a running average of their frame length adjustments. When a node computes an adjustment based on the timing data from at least two received messages, it will add this adjustment to the running total of all recorded adjustments, and increment the adjustment counter. Whenever a node receives no messages, it will divide the total sum of recorded adjustments by the number of adjustments in order to compute a historical average.

We have also implemented an *oracle* version of our history improvement. This version assumes that a node can determine its drift rate with great precision, allowing the node to use its exact clock frequency offset to make adjustments when isolated, rather than relying on an approximation. This version of history, called *histOracle*, will show us how effective the history optimization could be, if we had a very accurate method of estimating drift. A node implements this behavior by computing δ_{ticks}, the difference in ticks counted by a perfect clock in T_{frame} and ticks counted by its clock in T_{frame}. In each frame that a node receives no messages, it will add δ_{ticks} to a cumulative offset, Σ_{ticks}. If $\Sigma_{ticks} > 1$, the node will adjust its local frame length by $\lfloor \Sigma_{ticks} \rfloor$ and subtract this adjustment from the accumulated total.

Pseudo-code listings for these two algorithms are provided in Section 5.5.

5.1.2 Detection

We have devised a new improvement for active detection, called *targeted **join** messages*. Normally, if a node in a superior syncgroup, A, detects (hears a **join** message

from) a node in an inferior syncgroup, B, it will simply ignore this message, assuming that the sender will eventually detect the existence of syncgroup A. Since, as discussed above, the detection probability is quite low and detection events are relatively rare, we should try to take advantage of A's detection of syncgroup B even if A is superior to B and therefore the node will not decide to merge. We can do this by allowing the detecting node in A to try to *target* syncgroup B's active period with its next **join** message. Using the timing details from the sender's message, the receiver can determine an offset between the two syncgroups and thus can estimate when B's next active period will begin. By sending the next **join** message in a slot that has a strong possibility of overlapping with B's active period instead of in a random slot, the node from syncgroup A greatly increases the chance of a neighboring node from syncgroup B to detect it.

5.1.3 Decision

In Section 4.2.3, we discussed the addition of *cluster tags* as a way of solving the decision problem. In that section we explained that a cluster tag C is composed of two integers, an *ID* and an *epoch*, and written $C = \{C.id, C.epoch\}$. The reason for the epoch is because node mobility will complicate matters. One can imagine that the unsynchronized network presented in Figure 2.2a has converged and all nodes have merged into the same syncgroup, sharing a common cluster tag as well. After some time, due to mobility, the nodes may have changed their locations and are now physically separated into two different subnets as shown in Figure 2.2b. It is likely that the median clock frequencies in the two subnets are different. Thus over time, the physically separated nodes, though once synchronized, will slowly drift apart. The nodes in each subnet should continue to maintain synchronization within their subnet, however, a problem can later arise because both of these groups have the same cluster tag but will no longer be in the same syncgroup. If later in the scenario mobility brings the two subnets into contact again, nodes in each syncgroup will ignore **join** messages from the other syncgroup, since neither possesses a *superior* cluster tag. This scenario is what we call a *cluster split*.

Our solution to this problem is to introduce *epochs*. Epochs are given precedence over IDs in implementing the $\succ$ relationship, so a tag with a higher epoch is always superior to a tag with a lower epoch. This essentially invalidates tags with lower (older) epochs when a higher (newer) epoch is created. We call this modified relation $\succ_e$: $A \succ_e B$ if and only if $((A.id > B.id)$ AND $(A.epoch == B.epoch))$ OR $(A.epoch > B.epoch)$. Using the epoch counter in its cluster tag, a node can increase the weight of its cluster tag by incrementing its epoch when detecting a cluster split. That is, when a node X with cluster tag $A = \{a, e\}$ detects another node Y also claiming cluster tag A but with a different active period, node X generates a new cluster tag $A' = \{a', e+1\}$. This new tag will contain a randomly generated ID, a', and an epoch counter one higher than that of the old tag, $e+1$. The higher epoch makes this new cluster tag superior and ensures that the node's synchronized neighbors will adopt and disseminate this new tag. Nodes generate a new random ID in order to prevent the epoch counter running up to infinity without resolving the split in the case that nodes from two syncgroups experiencing a split of cluster tag $B = \{b, e\}$ independently and simultaneously detect

each other. If they did not generate a new ID, each node would increment the epoch of its cluster tag but keep the same cluster ID, resulting in both nodes having the same tag, $B' = \{b, e+1\}$. This process could repeat indefinitely, causing the epoch counter to count upwards to infinity without resolving the cluster split. Note that because epochs are implemented as a simple 8-bit integer, the epoch counter will roll over to zero after 256 epochs have been created. This roll over can be handled gracefully by using a modified comparison operation that treats epochs in the first quarter of the range (i.e., 0..63) as greater than epochs in the last quarter of the range (i.e., 192..255). Because we try to avoid creating unnecessary epochs by using randomly generated IDs, 8 bits should provide enough values for our simulations. There could be scenarios where more epoch values would be required, but this does not affect the general concept of our solution.

5.1.4 Notification

In this chapter we do not evaluate any *new* notification functionality, but rather re-evaluate the notification behavior described in Section 4.2.4.

5.2 Experimental setup

The primary goal of this research is to evaluate the performance of the standard GMAC synchronization, as well as a number of behavioral modifications that we have previously explained. We will perform our evaluation by simulating networks of various sizes, densities, and mobility patterns.

The novel contribution of this work is an evaluation of how the protocols handle various types of mobility. As discussed throughout the chapter, GMAC relies on local decision making, and thus the performance will depend greatly on the number of neighbors that nodes have (i.e., their degree), as well the quality of the wireless links between those neighbors. We use two methods of varying the network topology and connectivity. First, we run simulations using several different static and mobile scenarios. Second, we vary the simulated transmission power of all nodes in order to alter the degree of the nodes for a given topology. Finally, we vary the total number of nodes and size of the network while keeping the nodes per square meter ratio, or *node density*, constant. This is important, because it allows us to directly vary the average number of neighbors per node (the average node *degree*) and overall network connectivity without changing the physical topology of the nodes.

5.2.1 Simulator parameters

Clocks Here we investigate GMAC's synchronization maintenance and group merging in the face of mobile nodes. As demonstrated in Chapter 4, GMAC's synchronization maintenance performs adequately across a wide variety of clock drift settings. As such, in this chapter we keep the *MaxClockDrift* fixed at the default setting of ±20 ppm and focus our experiments on other parameters.

Table 5.1: Mobile topologies investigated in this chapter

Nodes	Dimension (Dm x Dm)	Area (m^2)	Node Density ($\frac{nodes}{m^2}$)
100	316	99,856	≈ 0.001
500	707	499,849	≈ 0.001
1000	1000	1,000,000	0.001
4000	2000	4,000,000	0.001

Network topology To better assess the strengths and weaknesses of the various configurations, we investigate the effect of topology on group merging. In particular, we look at network size and node mobility. We investigate a number of mobile scenarios, created using the BonnMotion[1] framework. We selected three (*Gauss-Markov*, *Random Walk*, and *Reference-Point Group Mobility*) of the fourteen mobility models provided by BonnMotion and generated traces of several sizes (shown in Table 5.1, with detailed parameters provided in Table 5.2). Notice that in all cases we ensure that the scenarios have an average node density of one node per thousand square meters, in order to increase comparability between results from various models. With the same motivation, we ensure that all mobility traces are generated with the same parameters (e.g., node speed $0 < s \leq 5$) whenever possible. For each type of mobility trace that we investigate, we extract the starting position of all nodes in that trace in order to create a static topology to compare against. That is, for a given mobility type (e.g., random walk) and size, we will look at two versions of the topology. The first is the mobile version, where each node's starting position and subsequent positions is determined by the specified BonnMotion mobility trace (e.g., called *mobile random walk*). The second is a static topology where nodes do not move from their starting position in the specified BonnMotion mobility trace (e.g., called *static random walk*). Though it may sound strange to discuss a "static" random walk topology, this naming allows us to appropriately describe the distribution of nodes in a given static topology. Furthermore, this allows us to isolate the effects of mobility by simulating networks with *identical* initial conditions, with the only difference being the movement of nodes in the mobile version.

Transmission density As discussed previously, the transmission density is the node density ($\frac{nodes}{m^2}$) × transmission area (m^2). All topologies studied in this chapter have the same node density, $\frac{1}{1000}\frac{nodes}{m^2}$. Thus, the transmission density will in all cases be equal to the transmission area divided by one thousand. In Table 5.3, we present the simulated transmission powers we use, along with the associated transmission range, transmission area and transmission density for each.

5.2.2 GMAC configurations

As in Chapter 4, we will analyze several distinct combinations of GMAC behaviors, called *configurations*. We have included a comprehensive list of our proposed behav-

[1] `http://net.cs.uni-bonn.de/wg/cs/applications/bonnmotion/`

Table 5.2: BonnMotion Parameters

model	GaussMarkov	RandomWalk	RPGM
randomSeed	1299022770517	1299023102669	1299023208481
x (m)	1000.0	1000.0	1000.0
y (m)	1000.0	1000.0	1000.0
duration (s)	10800.0	10800.0	10800.0
nn (nodes)	1000	1000	1000
circular	false	false	false
maxspeed (m/s)	5.0	5.0	5.0
minspeed (m/s)	N/A	0.1	0.1
maxpause (s)	N/A	60.0	60.0
updateFrequency (s)	2.5	N/A	N/A
angleStdDev (rad)	0.39269908169	N/A	N/A
speedStdDev (m/s)	0.5	N/A	N/A
bounce	true	N/A	N/A
initGauss	false	N/A	N/A
uniformSpeed	true	N/A	N/A
mode	N/A	t	N/A
modeDelta	N/A	60.0	N/A
groupsize_E	N/A	N/A	12.0
groupsize_S	N/A	N/A	2.0
pGroupChange	N/A	N/A	0.1
maxdist	N/A	N/A	25.0

Table 5.3: Transmission power settings and associated transmission densities

TX Power (mW)	TX Range (m)	TX Area (m^2)	TX Density (nodes)
0.005764	8.920	249.965	≈ 0.25
0.016303	12.615	499.947	≈ 0.50
0.046111	17.841	999.973	≈ 1.00
0.084712	21.850	1499.867	≈ 1.50
0.130423	25.231	1999.948	≈ 2.00
0.182271	28.209	2499.915	≈ 2.50
0.239602	30.901	2999.818	≈ 3.00
0.368891	35.682	3999.892	≈ 4.00
1.043381	50.462	7999.794	≈ 8.00
1.916814	61.803	11999.661	≈ 12.00
5.421568	87.403	23999.520	≈ 24.00
15.334513	123.607	47999.422	≈ 48.00

ioral modifications in Table 5.4

- <***Active***> The default GMAC behavior, as described in Section 2.4.

Table 5.4: New GMAC Behaviors

Name	Abbreviation	Synchronization Aspect	Section
Active Detection	A	Detection	5.1.2
Passive Detection	P	Detection	5.1.2
Listen before Merge	L	Detection	5.1.2
Target **join** messages	T	Detection	5.1.2
Cluster tags	C	Decision	5.1.3
Notify on Merge	N	Notification	5.1.4
History	H	Maintenance	5.1.1
HistOracle	O	Maintenance	5.1.1

- <***Active+History***> This configuration uses both active detection and our *history* optimization described in 5.1.1.
- <***Active+histOracle***> This configuration uses active detection combined with our *oracle* history method described in 5.1.1.
- <***Passive***> The default GMAC behavior, using passive detection in place of active detection with p_l =0.17%, as described in Section 4.2.2.
- <***Active+Cluster***> Same as <*Active*>, but using cluster tags (Section 4.2.3 and Section 5.1.3in order to make consistent merge decisions.
- <***Active+Cluster+Notify***> The same as <*Active+Cluster*>, but nodes do not immediately merge into a newly discovered group, rather they wait one frame in order to send merge messages (Section 5.1.4).
- <***Active+Cluster+Listen***> The same as <*Active+Cluster*>, but nodes do not immediately merge into a newly discovered group, rather they listen for a whole frame in order to discover the best cluster tag in range (Section 5.1.2).
- <***Active+Cluster+Notify+Target***> This configuration combines active detection, cluster tags, merge messages, and targeted **join** messages. The idea of targeting **join** messages was introduced in Section 5.1.2.
- <***Active+Cluster+Notify+Target+histOracle***> This configuration makes use of active detection and combines all of our modifications that have shown improvement in synchronization behavior: cluster tags, merge messages, targeted **join** messages, and our oracle history method.

5.3 Evaluation

We evaluate the performance of GMAC and our various modifications in the same way that we have described synchronization throughout the thesis. We begin by looking at the maintenance of synchronization by simulating networks composed of nodes that start in the `SYNCHRONIZED` state. We will then evaluate the performance of three aspects of merging through a number of different scenarios including asynchronous initializations of mobile topologies. As mentioned earlier, all results in this

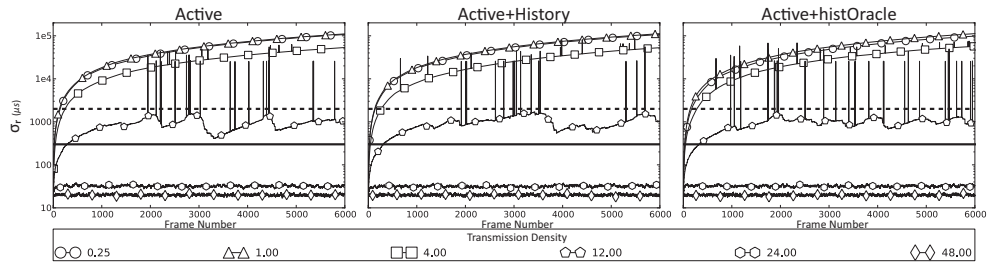
section are generated by simulating each combination of transmission density GMAC configuration eight times, each iteration with a different random seed. The lines representing each density setting for a particular GMAC configuration represent an average across these eight repetitions of the same simulation. Unless otherwise noted, all results are present in logarithmic scale on the y-axis. We do this because the timing differences between mostly synchronized nodes are on the order of tens to hundreds of microseconds, while the differences between unsynchronized nodes will range up to half the frame time (i.e., five hundred thousand microseconds). Plotting the results on a logarithmic scale allows us to see the behavior of the network more clearly.

5.3.1 Maintenance

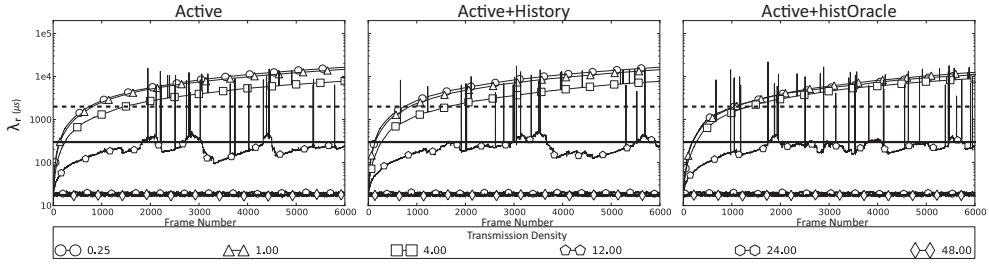
Maintenance of a common active period amongst synchronized groups of nodes is the foundation of complete network synchronization. It would be pointless for all nodes to converge on a shared active period if they would drift apart a few frames later. Synchronization maintenance depends upon periodic communication with other nodes (from the same syncgroup). A node may go many frames without communication and remain sufficiently synchronized, but eventually nodes must exchange messages with timing information in order to compensate for their drift. How long a node can be isolated and still remain synchronized depends on the magnitude of the frequency offset in its internal clock. In static topologies, it would be unlikely that a node is not able to communicate with one of its neighbors for an extended period of time. Even a wireless link with a failure rate of 90% would provide sufficient message exchange for nodes to stay synchronized with their neighbors. Mobile topologies, on the other hand, could easily expose nodes to extended periods of isolation.

We evaluate how well GMAC maintains synchronization between nodes by simulating a network that is initially synchronized. That is, all nodes begin execution at the same time in the `SYNCHRONIZED` state. We start our simulations with static topologies, then we look at what happens when nodes are allowed to move. In order to evaluate synchronization maintenance, we look at the standard deviation of frame start times, σ_r. The simulations begin with all nodes perfectly synchronized, so σ_0 will be 0. The various random frequency offsets assigned to each node cause σ_r to gradually increase as the simulation progresses. The median algorithm should prevent the nodes from drifting too far apart, and the standard deviation of the reported times shows us to what extent it is successful.

The three GMAC configurations we will examine here are: <*Active*> (*A*), <*Active+History*> (*AH*), and <*Active+histOracle*> (*AO*). We evaluate the performance of each of these configurations by looking at the difference (standard deviation) in the per-frame start times reported by all simulated nodes, σ_r, as described in Section 4.1.3. In each graph, we plot the frame number on the x-axis and the standard deviation on the y-axis. We have included a dashed horizontal line at $\sigma_r = 2ms$, or about three TDMA slots. Note that this is the same bound we use for ϵ when calculating syncgroups. This line represents a rough approximation of the permissible amount of timing variability that can still be considered synchronized. That is, per-frame standard deviations below that line indicate that the entire network is loosely synchronized and the majority



(a) Global standard deviation of start times, σ_r



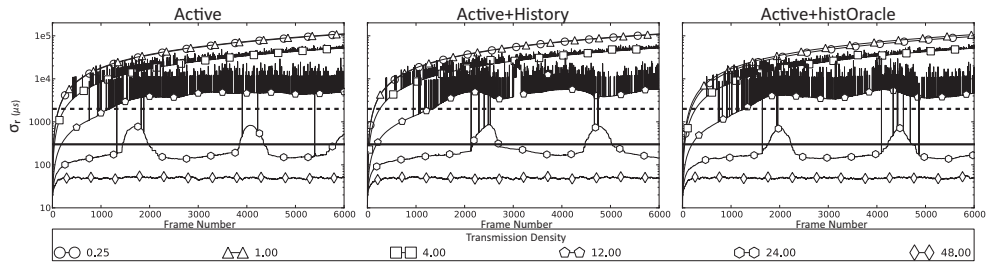
(b) Local (1-hop) standard deviation of start times, λ_r

Figure 5.1: The performance of GMAC's synchronization maintenance on a static 100-node Random Walk network

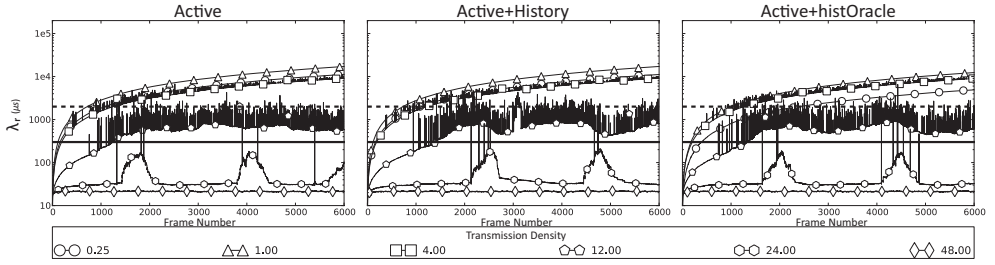
of nodes will be in the same syncgroup. Per-frame standard deviations significantly higher than this line, however, indicate that the entire network is not in a synchronized state and it is likely that several distinct syncgroups exist. Network wide synchronization with $\sigma_r < 300\mu s$ (about 10 ticks, or one third of a slot) indicates very tight global synchronization and a fully converged state. We draw a solid horizontal line on the graphs to represent this.

In Figure 5.1, we see the performance of our three test configurations on one of our small static topologies. The upper set of graphs shows the global standard deviation of start times (σ_r), while the lower set shows the average local standard deviation (λ_r). In this case, we look at a static random walk deployment of one hundred nodes. We can immediately see that at the lower densities the initially perfect synchronization is not maintained. In fact, only the two highest transmission densities, 24 and 48 nodes, provide for high-quality synchronization. At a density of 12 nodes the network is very tenuously connected, and the sparse topology results in synchronization right on the borderline of what we consider acceptable. At the three lowest densities, we see that the overall connectivity is too low to provide sufficient message exchanges to maintain complete network synchronization. At such low densities the network is split into several disjoint subnets, forcing the network to gradually devolve into disjoint syncgroups as nodes drift apart. Notice in the results for low densities that the initially perfect synchronization is lost within at most three to four hundred frames. The most clear result is that neither the <*Active+History*> nor the <*Active+histOracle*> configurations perform significantly better than the standard <*Active*> configuration in this scenario.

In Figure 5.2, we test our protocols on a much larger static random walk topology, this time with one thousand nodes. Increasing the scale of the network by an order of



(a) Global standard deviation of start times, σ_r



(b) Local standard deviation of start times, λ_r

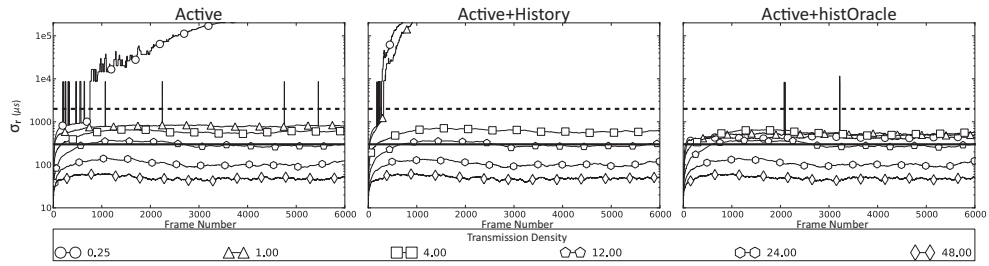
Figure 5.2: The performance of GMAC's synchronization maintenance on a static 1000-node Random Walk network

magnitude does not change the results by much. In fact, they look remarkably similar. In the larger network, however, a density of 12 is no longer sufficient for GMAC to maintain synchronization, regardless of any attempts to compensate for drift while isolated (seen in the two plots on the right). The larger network size means that there can be more and larger sparse regions in the topology, balanced by regions of higher density elsewhere. We can also see from the results that, even at a transmission density of 24 nodes, GMAC struggles to keep all one thousand nodes synchronized. It seems that, at least for static topologies, the two variations on history maintenance offer little benefit. In these experiments, it is only changing the transmission density that has any significant effect on the results.

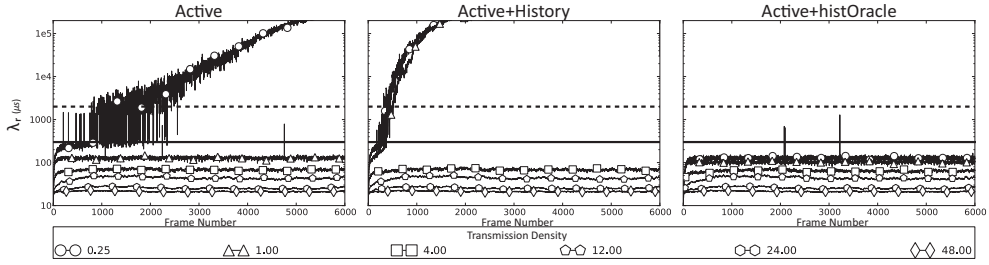
5

Finally we look at GMAC's behavior in a network of mobile nodes in Figure 5.3. The difference in the performance between static and mobile topologies is striking. It is clear that GMAC achieves far superior results on a dynamic network topology compared to static topologies. With the addition of mobility, we also see a significant change in performance based on the selected GMAC configuration. The basic <*Active*> configuration on the left maintains the initial synchronization for all tested transmission densities, with the exception of the lowest setting of 0.25.

We can see that <*Active+History*> actually hurts synchronization maintenance. In sparse networks nodes will be isolated more often, and thus they will rely on the history-based approximation of their clocks' frequency offset in order to try to stay synchronized. However, in such networks nodes will also receive far fewer messages upon which to base that approximation. One can imagine that a few frame length adjustments that run contrary to the node's actual clock frequency offset will "pollute" the node's history. This will, in turn, lead the node to make a drift approximation



(a) Global standard deviation of start times, σ_r



(b) Local standard deviation of start times, λ_r

Figure 5.3: The performance of GMAC's synchronization maintenance on a mobile 1000-node Random Walk network

that is not representative of the node's true drift. Later timing adjustments based on this erroneous approximation will cause the node to desynchronize with its group *faster* than it would without making any adjustments at all. It is clear, however, that the fault lies with our approximation algorithm, since neither the <*Active*> nor the <*Active+histOracle*> configurations suffer from this behavior.

Finally, the results for the <*Active+histOracle*> configuration, show a clear benefit over <*Active*>. The three lowest density settings maintain good synchronization, and perform almost identically with this enhancement. We can see that this benefit is limited to the lower density experiments, and the higher density experiments (where nodes are very rarely isolated) show almost identical performance with <*Active*>. What this tells us is that making timing adjustments in isolation based on an approximation of local clock frequency offset could be a strong addition to the protocol, if we can find a cheap and easy method of approximating a node's drift.

As seen in Figures 5.1, 5.2, and 5.3, the differences between σ_r and λ_r are the magnitude of the deviation, not the behavior over time. From the above results, we can clearly see that the global standard deviation follows the same pattern as the local standard deviation. This is expected, and for this reason, we present only the global standard deviation (that is, σ_r) from now on.

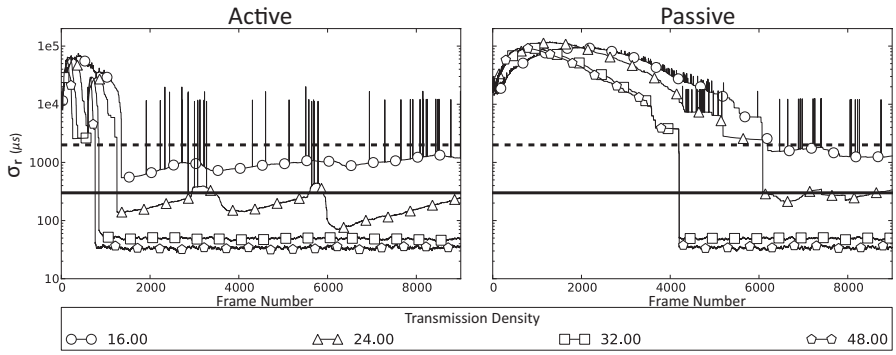
5.3.2 Detection

As we have chosen complete network convergence as our goal, we should ensure that detection of other synchronized groups happens quickly, but also with as little energy

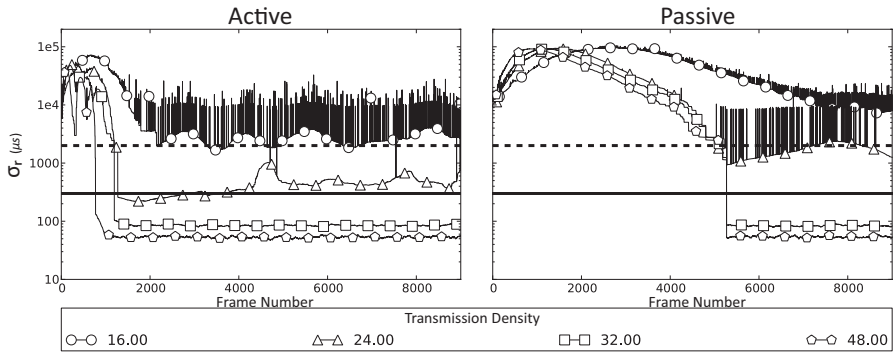
expenditure as possible. Here we evaluate both the active and passive methods of detection by artificially creating two synchronized groups. We are interested in only two syncgroups because we want to eliminate the possibility of cycles in GMAC's timing-based decision relation, $\succ_t$ (Section 5.1.3), obscuring our evaluation of the detection mechanism. The first syncgroup is composed of a single node, node 0, while the second syncgroup is composed of the rest of the nodes in the simulated network. Both groups start, independently, in the `SYNCHRONIZED` state. The large group begins executing at $t = 0.5s$, while node 0 begins at a random time $0s < t \leq 1s$. We choose a random start time for the singleton group in order to vary which group is superior according to $\succ_t$. In runs where node 0 starts up *before* the synchronized group, the large group will be forced to merge with node 0. In contrast, runs were node 0 starts *after* the other nodes, node 0 must detect the other group and merge with them. As above, we perform our experiments using the same variety of static and mobile topologies, and at several different transmission power settings. Our main metric to evaluate detection will again be σ_r.

The two GMAC configurations we will examine here are: <*Active*> (A) and <*Passive*> (P). Both configurations succeed in synchronizing the 100-node topologies, both static and mobile, so we do not present those results here. We instead look at the 500-node and 1000-node networks. In Figure 5.4, we present the results of the static topologies. We see in the left sides of Figures 5.4a and 5.4b that using active detection quickly leads to tight synchronization at the three highest density settings, but can achieve only loose synchronization at the lowest investigated density. Clearly density is a determining factor in not only whether or not synchronization will succeed, but also in what bounds can be achieved. However, we do see some conflicting results regarding density for the case of passive detection. In the right side of Figure 5.4a the results of transmission density 48 are inferior to the results of lower density settings 24 and 32. Similarly, in the right side of Figure 5.4b, a density setting of 32 nodes leads to worse performance than at a density of 24. In very dense networks with most nodes mutually synchronized, the common active period will be heavily loaded with messages. In such situations message collisions will be common, so there is a chance that a node performing passive detection will not receive any messages during its inactive period. Because listening during the inactive period is probabilistic and has a low probability of success ($< \frac{1}{600}$), missing an opportunity to detect another syncgroup can be quite costly and delay synchronization for hundreds or thousands of frames. The active detection method is significantly more effective at the same energy cost, synchronizing the network within two thousand frames. Passive detection's best performance is over 4000 frames, seen in the static 500-node network at transmission density 32.

As in our evaluation of synchronization maintenance, active detection also performs significantly better in the case of mobile topologies, seen in Figure 5.5. As nodes move around the simulated area, they directly exchange messages with a much larger number of nodes than in a static scenario. This allows synchronization information to propagate via physical movement as well as by radio communication, and lends a strong performance benefit. Furthermore, the effects of transmission density are less pronounced in the mobile scenarios. Using active detection (left side of Figures 5.5a and 5.5b), GMAC synchronizes both the 500- and 1000-node networks within about 1000 frames. Passive detection (right side of Figures 5.5a and



(a) Static Random Walk network, 500 nodes



(b) Static Random Walk network, 1000 nodes

Figure 5.4: Evaluating GMAC's detection mechanisms in static networks

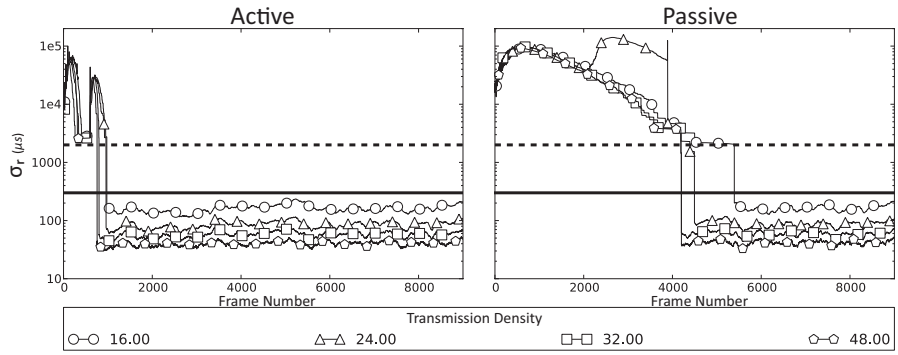
5.5b), while also successful, takes four to six times as long to synchronize the same network. Finally, mobility has also remedied the strange passive detection behavior where higher density topologies can take longer to synchronize than lower density ones.

5

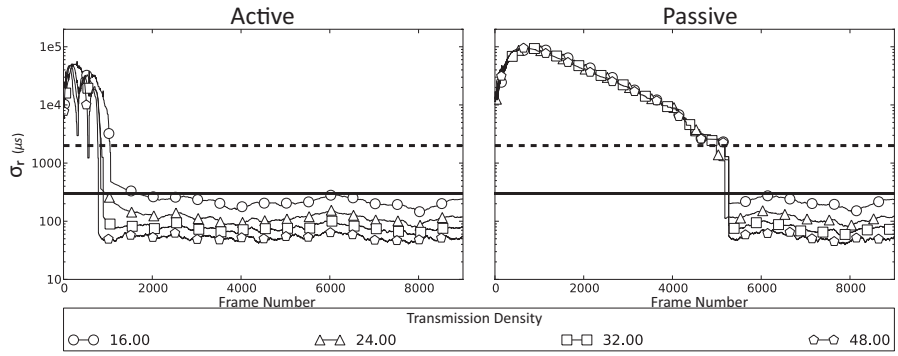
5.3.3 Decision

To examine the *decision* aspect of merging, we study networks under an asynchronous initialization. In these simulations, we create the conditions for a chaotic network start. All nodes start unsynchronized and must initially *detect* their neighbors in order to form local syncgroups. The better syncgroups, determined by the relation $\succ$, will continue to grow in size as nodes discover neighbors in superior syncgroups and *decide* to merge with them. By measuring what percentage of the nodes have synchronized (to a common active period) as the simulation progresses, we can see the effects of deterministic decisions made by the nodes. For each frame, we count the percentage of synchronized nodes as described in Section 4.1.3. The GMAC configurations we will examine here are <*Active*> (A) and <*Active+Cluster*> (AC).

As we are interested in networks of large scale, we focus our evaluation of decision mechanisms on mobile 1000-node topologies. We present, on the left side of



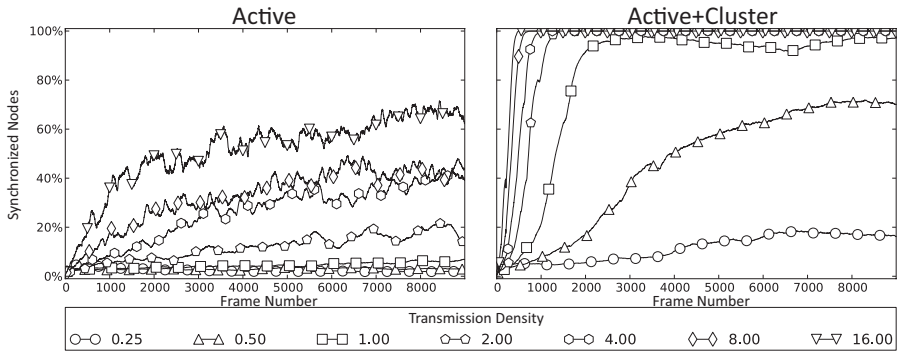
(a) Mobile Random Walk network, 500 nodes



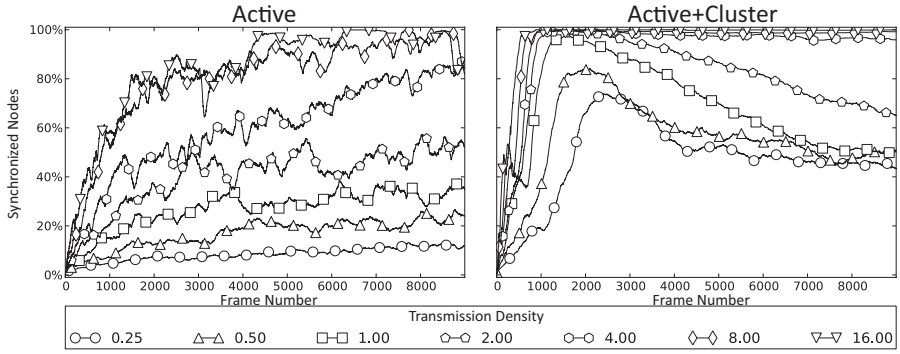
(b) Mobile Random Walk network, 1000 nodes

Figure 5.5: Evaluating GMAC's detection mechanisms in mobile networks

Figure 5.6, the performance of GMAC's default decision relation, $\succ_t$. On the right side of Figure 5.6 are the results using the improved relation $\succ_c$, based around cluster tags. Without cluster tags, the number of syncgroups that form during an asynchronous initialization virtually guarantee that there will exist cycles in the network. This is evidenced by the fact that the <*Active*> configuration fails to consistently synchronize 100% of the nodes at any transmission density in the case of Gauss-Markov mobility, and barely manages to achieve 100% synchronization at the highest transmission density with reference-point group mobility. <*Active+Cluster*>, however, quickly converges the entire network to a single active period at all but the lowest transmission densities in Gauss-Markov topologies. It does not perform quite so well on the reference-point group mobility trace, but this pattern lends itself to physically isolated groups of nodes. Still, the addition of cluster tags does allow GMAC to perform far better than without. It is also worth noting that the cluster tags provide for extremely consistent performance as well, as seen by the sharp slope in the first thousand frames. This indicates that the eight runs performed very similarly, whereas the results without cluster tags show shallower (or even horizontal) lines.



(a) Mobile Gauss-Markov network, 1000 nodes



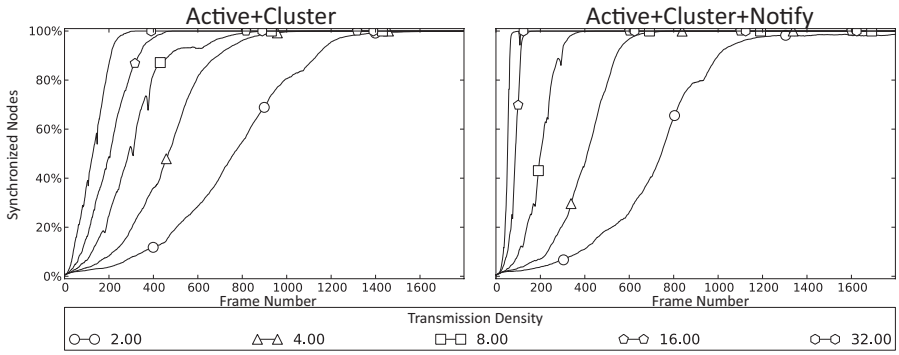
(b) Mobile Reference Point Group network, 1000 nodes

Figure 5.6: Evaluating GMAC's decision mechanisms in mobile networks

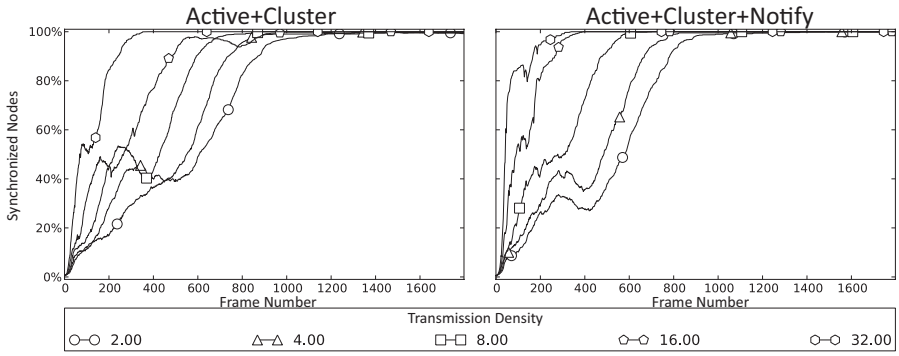
5.3.4 Notification

To examine the *notification* aspect of merging, we will again make use of asynchronous initialization scenarios. We look at simulations similar to those as above, but this time with an eye to the performance of the merge messages optimization. In keeping with our goal of complete network convergence, we will focus on the percentage of synchronized nodes, as before. The two GMAC configurations we will examine here are: <*Active+Cluster*> (*AC*) and <*Active+Cluster+Notify*> (*ACM*).

In Figure 5.7, we see the performance of GMAC's default behavior without merge messages on the left side, and the behavior with merge messages on the right side. The top two graphs depict the random walk mobility pattern we have seen throughout this section. The addition of merge messages allows the network to converge in an average of just over 100 frames, while the configuration without messages takes more than three times as long. It is interesting to note that, with merge messages, the percentage of synchronized nodes becomes almost vertical line at high densities. This is because the merge messages greatly reduce the chance of nodes being "left behind" as the other nodes in their group detect a better group and merge into it. This effect is seen in the much more gradual slope of the results without any notification method on the left. The effect is also strongly tied to a node's transmission range, as the results



(a) Mobile Random Walk network, 1000 nodes



(b) Mobile Reference Point Group network, 1000 nodes

Figure 5.7: Evaluating notification mechanisms in mobile networks

for lower densities reflect little improvement. The bottom graphs show the behavior on nodes following the reference-point group mobility pattern. This mobility trace keeps nodes in tight physical groups that move in reference to a common point. The effect on the synchronization behavior is clear, as this pattern restricts interactions between nodes to mainly those in the same reference point group. Only when these groups cross paths are there opportunities for synchronization information to pass between them. The results for this mobility model also show a strong correlation with the network density, as the effect of the merge messages is more pronounced at high density.

5.3.5 Detection revisited

Two of our detection optimizations, listen-before-merge and targeted **join** messages, should be far more effective if combined with cluster tags. For that reason, we revisit the issue of detection here, shown in Figure 5.8. This time, however, we instead use an asynchronous start, in order to allow for more syncgroups to be established. We also investigate lower transmission densities than in the previous detection study, with the same aim. The larger number of syncgroups will provide more opportunity for the configurations we examine here to demonstrate their utility. Both optimizations require that there be other syncgroups in a node's vicinity in order to see any

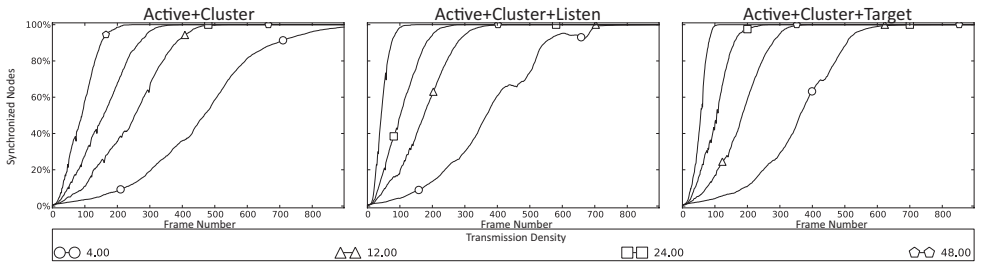
benefit. We study the behavior of <*Active+Cluster*> (*AC*), <*Active+Cluster+Listen*> (*ACL*), <*Active+Cluster+Target*> (*ACT*) to see whether we can improve syncgroup detection even further. We have simulated the performance of these three GMAC configurations under each of our three mobility patterns, though we leave out the results for Gauss-Markov as they are very similar to those from the random walk trace. In the leftmost plot of the figures we again see the performance of the <*Active+Cluster*> configuration. For comparison, we present the results of the <*Active+Cluster+Listen*> configuration in the center plot, and the results of <*Active+Cluster+Target*> on the right.

We can see that the targeted **join** messages offer an observable performance benefit, particularly at high transmission density. The benefit is most pronounced in the Gauss Markov mobility model. The reason for this is that this optimization balances out the asymmetric decision behavior. That is, nodes would normally ignore messages from inferior clusters. This means that effectively only an inferior group can detect a superior group, since a superior group will always ignore its messages. Targeting **join** messages allows us to effectively double the detection probability by making the process symmetrical. As we have seen throughout this section, the effect of the optimizations is limited by the density of the network. The better connected the network topology, the less benefit afforded by the targeting. At the highest transmission density, 48, we do see an approximate doubling of performance, from two hundred frames with <*Active+Cluster*> to one hundred frames with <*Active+Cluster+Notify+Target*>. The results for the reference point group mobility (Fig. 5.8b) also show a stronger performance increase at high density when using the targeting behavior. However, the performance benefit quickly diminishes or disappears entirely at lower densities.

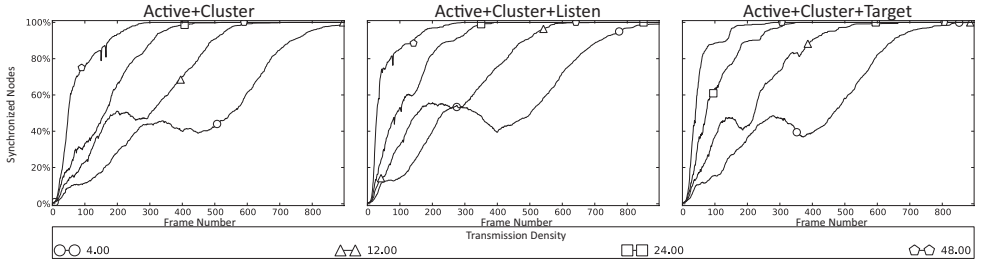
We find the results of the listen before merge optimization to be disappointing, especially considering the energy cost of this behavior. Listening for an entire frame costs about the same as sending 600 **join** messages, quite expensive indeed. The performance is comparable, but inferior, to that provided by targeted **join** messages. However, the targeting does not imply any additional radio time and thus little to no additional energy cost, making it the superior choice.

5.3.6 Larger networks

As we have emphasized several times, our chief interest is scalability. As sensor nodes continue to fall in price, very large-scale networks will become economically feasible. Thus, we end the discussion of our simulated results focusing in that direction. The only significant difference between these experiments and those described earlier are the number of simulated nodes. We still look at the same mobility patterns, but here we observe the behavior of 4000 nodes. We will again look at an asynchronous initialization, since this type of scenario presents a worst-case for network-level synchronization. Because running and processing these simulations are quite demanding, we look at only three transmission density settings (2, 8 and 32) and two GMAC configurations (<*Active+Cluster*> and <*Active+Cluster+Notify+Target*>).



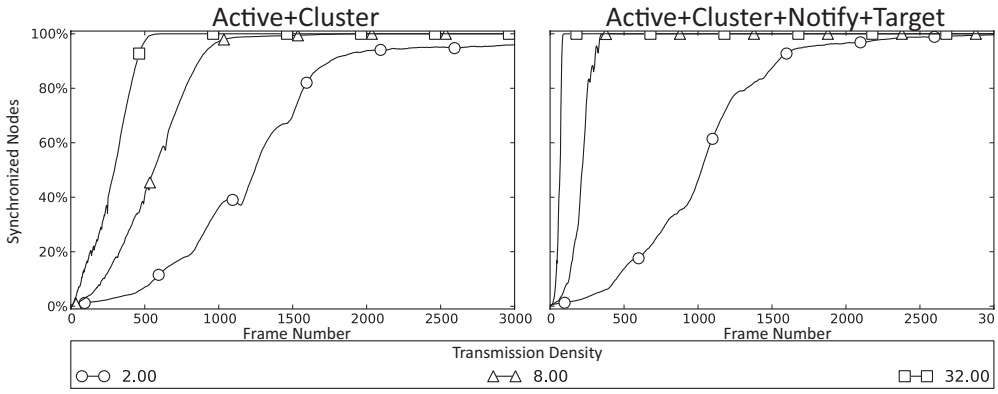
(a) Mobile Gauss-Markov network, 1000 nodes



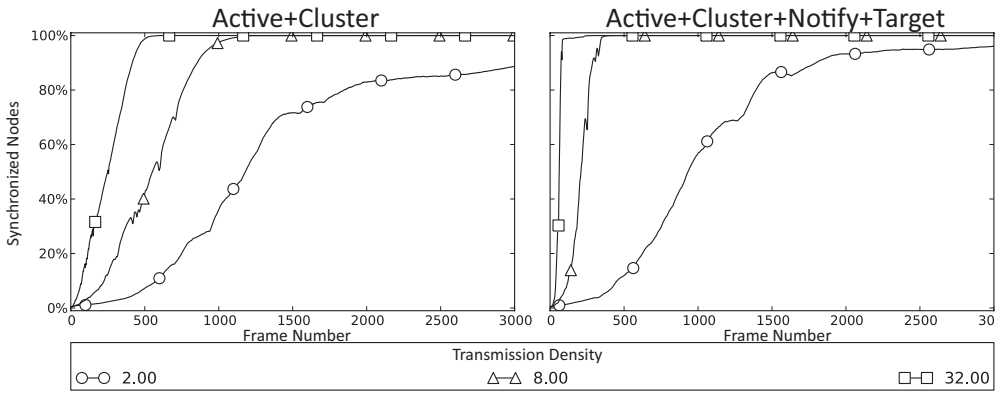
(b) Mobile Reference Point Group network, 1000 nodes

Figure 5.8: A look at the performance of further proposed improvements to GMAC's detection behavior, 1000-node mobile networks

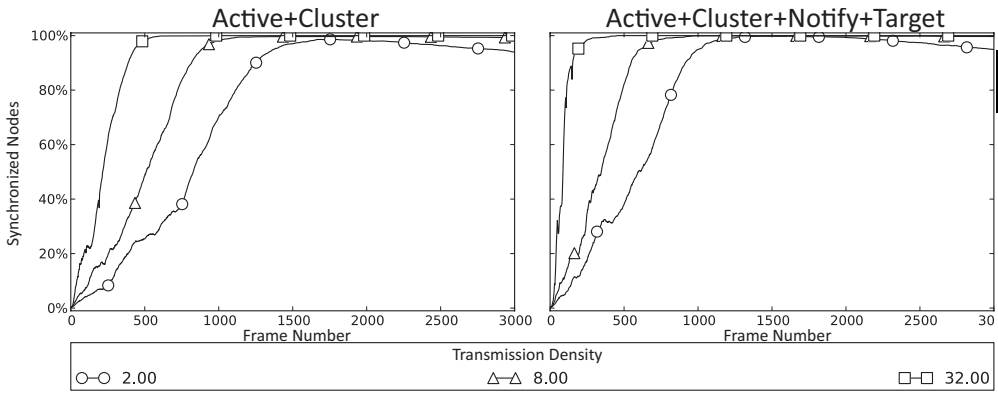
In Figure 5.9, we present results for all three mobile topologies. We can see that the results for the Gauss-Markov mobility pattern (Fig. 5.9a) and those for random walk (Fig. 5.9b) are very similar. Both of these mobility patterns lead to network topologies with relatively uniform node density. This can be seen in the results, as the percentage of synchronized nodes trances a smooth line. The results of the reference point group mobility pattern (Fig. 5.9c) present a more bumpy and irregular pattern. As mentioned previously, this type of mobility manifests a much less uniform node density as groups of nodes move together around their common reference points. Nevertheless, both tested configurations manage to converge all three simulated deployments at transmission densities 8 and 32. The uniformity of the Gauss-Markov and random walk topologies provide the best setting for the <*Active+Cluster+Notify+Target*> configuration to outperform the simpler <*Active+Cluster*> configuration. The left-hand graphs show the behavior of <*Active+Cluster*>, which reaches 100% synchronization in about five hundred frames at density 32 and one thousand frames at density 8, on both topologies. The right-hand side shows that the addition of merge messages and **join** message targeting reduces the time required to about 80 and 400 frames, respectively. Similar, though less dramatic, improvement is seen with the reference point group mobility pattern as well, but GMAC's performance is limited by the lower connectivity of the network. Finally, we can see that our proposed improvements make little if any difference at the lowest examined transmission density, regardless of the topology.



(a) Mobile Gauss-Markov network, 4000 nodes



(b) Mobile Random Walk network, 4000 nodes



(c) Mobile Reference Point Group network, 4000 nodes

Figure 5.9: GMAC at very large scale: 4000-node mobile networks

5.4 Conclusions

The main contribution of this chapter is a thorough evaluation of MAC-level synchronization in ultra-low duty cycle, large scale, mobile (and static) networks. In order to perform this evaluation, we examined the two distinct subproblems of network-level synchronization through numerous simulations. The results of these simulations show that both problems *are* solvable, and GMAC can be used to achieve remarkably low duty-cycles, even with relatively inaccurate clocks. In addition, our solution uses energy at a fixed rate, which allows for very accurate predictions of network lifetime. And, perhaps most importantly, we have demonstrated that GMAC is not only capable of synchronizing *all nodes* in a network so that they share a common active period, but also doing so in a completely decentralized manner. Removing the need for special "cluster-head" nodes makes planning and deploying a sensor network simpler and less costly.

We investigated the maintenance of existing synchronization between groups of nodes, the first aspect of synchronization, by simulating networks of 100, 500, and 1000 initially-synchronized nodes. We varied the transmission range of all nodes in order to see the effects of overall network density on synchronization maintenance. We found that GMAC's simple median algorithm was able to maintain synchronization in a mobile 1000-node network with average densities as low as one node per transmission area. Using an oracle to estimate a node's drift allows GMAC to maintain synchronization in even sparser networks, demonstrated at a density of 0.25 nodes per transmission area. The results for the static versions of the same topologies were not as good. Static networks require a much higher transmission area (about 16 nodes per transmission area) in order to maintain synchronization, due to the network being partitioned into multiple subnets at lower levels. Furthermore, the results for the <*Active+histOracle*> configuration, show a strong benefit. Using this mechanism, nodes can maintain good synchronization at very low transmission density. At higher density (where nodes are very rarely isolated), the need to make timing adjustments in isolation disappears and adds no performance benefit. This tells us that making timing adjustments in isolation based on an approximation of local clock frequency offset could be a strong addition to the protocol, assuming we can find a cheap and easy method of approximating a node's drift over time.

The second part of synchronization, merging separately synchronized groups of nodes, has been split into three orthogonal subproblems. We investigate each of the three subproblems (detection, decision and notification) using different mobility patterns and starting conditions. We looked at two methods of detection, active and passive. In all investigated topologies and densities, active detection always outperformed passive detection by a significant margin. We attribute this to active detection's ability to detect multiple neighbors with a single broadcast, as described in 5.1.2. Additionally, we demonstrated that the combination of active and passive detection can offer small performance benefit, but will generally not outweigh the additional energy cost. Performance is even further increased by using our proposed technique of targeted **join** messages, effectively doubling the detection rate. Regarding merge decisions, our proposed cluster tags strongly improve the chances

of eventual synchronization. We have demonstrated their efficacy in mobile networks ranging from 100 to 4000 nodes, showing the importance of deterministic merging decisions. Finally, our proposal of using a header field for notifying neighbors of local merge decisions can drastically reduce the time for a network to reach a synchronized state, by as much as a factor of eight on our 4000-node topology. These small modifications to GMAC's current behavior radically increase its suitability for large scale mobile networks. The key insight is that as synchronized groups build up, the merge messages allow GMAC to leverage an inferior group's existing synchronization to rapidly merge *whole syncgroups*, not just individual nodes. Combined with the total ordering provided by cluster tags to solve the problem of which group to merge with, large and complicated networks can be synchronized in just a few minutes.

In the end, the most clear result of our investigations is that the achievable level of synchronization is dictated by two things. The first is transmission density, or the number of neighbors an average node will have. The second is whether or not the topology is dynamic. In both cases, the underlying issue is: how easily can the synchronization information be propagated throughout the entire network? With very low transmission densities, a node will have infrequent communication with other nodes which will greatly limit the opportunities for disseminating synchronization data. Conversely, high transmission density means that this data can move a great distance during each frame. Similarly, mobile scenarios generally facilitate network synchronization because the required information can be propagated faster. This occurs because nodes physically move around, carrying their data with them. Both of these factors, density and mobility, are aspects of the overall network topology. In the end, it is the network's topology that will determine whether synchronization will succeed.

Our next step will be to verify the performance of these modifications by implementing them to run on our existing hardware. We have executed a number of real-world experiments with the default GMAC protocol, though at a much smaller scale than we investigate here. It is difficult to directly compare these real-world experiments with our simulated results, because there is no authoritative time source with which to compare nodes' local timing data. Furthermore, the scale at which the basic GMAC protocol begins to show serious problems achieving synchronization (i.e., one thousand nodes and more) makes it expensive and very time-consuming to replicate some of our simulations. Nevertheless, we have a high degree of confidence that our simulated results on smaller networks qualitatively reflect the behavior of GMAC on the reference hardware. We will actively seek opportunities to further confirm the veracity of our simulations with real-world results.

5.5 History algorithms

In this section we present the details of the *history* and *histOracle* algorithms discussed in Section 5.1.1. In both cases, we show the algorithm's behavior across three functions: `Init()`, `PhaseCorrect()`, and `ComputeBlindCorrection()`. A node calls `Init()` when it starts up in order to properly initialize the history state. `PhaseCorrect()` is called each frame by a node to adjust the length of its local frame based on the computed phase error and number of neighbors (i.e., message receptions). Finally, `ComputeBlindCorrection()` is called by `PhaseCorrect()` if the node received no messages that frame. This function will try to determine a reasonable frame-length adjustment to make in the absence of any new timing information. We provide listings for the history algorithm in Algorithm 5.1, and for the histOracle algorithm in Algorithm 5.2 below.

Algorithm 5.1: The *History* Algorithm

```
Init()
    // Initialize the cumulative correction to zero
    cumCorrection = 0;
    // Initialize the correction count to zero
    numCorrections = 0;

PhaseCorrect(phaseError, numNeighbors)
    if numNeighbors == 0 then
        // If we have no neighbors, we must make a `blind'
           correction
        phaseError = ComputeBlindCorrection();
    else if numNeighbors > 1 then
        // If we have more than one neighbor, we will remember this
           correction
        cumCorrection = cumCorrection + phaseError;
        numCorrections = numCorrections + 1;
    // Finally, adjust our frame length based on our computed phase
       error
    AdjustFrameLength(phaseError);

ComputeBlindCorrection()
    correction = 0;
    // We should only make adjustments if we have enough history
    if numCorrections > MinCorrections then
        // Simply compute the running average of our past
           corrections
        correction = cumCorrection ÷ numCorrections;
    return correction;
```

Algorithm 5.2: The *HistOracle* Algorithm

```
Init()
    // Initialize the cumulative correction to zero
    cumCorrection = 0.0;
    // An `oracle' determines how many clock ticks are lost (or
       gained) per frame
    driftTicks = (1.0 - offset()) * frequency * FrameLength;

PhaseCorrect(phaseError, numNeighbors)
    if numNeighbors == 0 then
        // If we have no neighbors, we must make a `blind'
           correction
        phaseError = ComputeBlindCorrection();
    // Finally, adjust our frame length based on our computed phase
       error
    AdjustFrameLength(phaseError);

ComputeBlindCorrection()
    // By default, we make no correction
    correction = 0;
    // Increment our cumulative phase error by the number of ticks
       lost/gained per frame
    cumCorrection = cumCorrection + driftTicks;
    if cumCorrection ≥ 1.0 then
        // If we have drifted at least one tick ahead, correct it
        correction = ⌊cumCorrection⌋;
    else if cumCorrection ≤ −1.0 then
        // If we have drifted at least one tick behind, correct it
        correction = ⌈cumCorrection⌉;
    if correction ≠ 0 then
        // Remove any correction we make from the cumulative error
        cumCorrection = cumCorrection- correction;
    return correction;
```

5

6. Scalable epidemic applications

This thesis thus far has been concerned almost exclusively with the operation of GMAC at the MAC/network layer. Here we will focus instead on the application layer. An interesting application area is measuring activities in social communities [6] [46] [51], and utilizing social structures to understand and stop the spreading of infectious diseases [52] [53]. In research related to this thesis we are using wearable wireless sensor nodes to detect proximity among thousands of people, in an attempt to discover the overall dynamic structure (*topology*) of a crowd. In this chapter we will discuss a specific application that represents a class of applications that can be easily implemented atop GMAC's API, namely applications utilizing *epidemic* communication. This application, called **NetSize**, attempts to estimate the size of the network of which it is a member.

The social ad hoc networks we investigate will have a high diameter, tens or even hundreds of hops across. In such a situation, the local network conditions (e.g., node density, neighborhood size, RF interference) may vary significantly between different locations within the same network. This, in turn, implies that one MAC protocol or one set of MAC parameters will not be optimal for all nodes in the network. In this case, we must allow nodes to *self-adapt* to local conditions. As previously discussed, adaptivity is an important research goal in this work. In this chapter we explore the NetSize application, which could provide the input to such a self-adaptation mechanism. By combining an accurate estimation of the network size with a knowledge of the (approximate) transmission range, nodes can estimate the density of the network. Given this information, nodes could, for example, increase their duty cycle to avoid excessive message collisions in very dense regions. To be even more concrete, a reasonably accurate estimate of a node's 3-hop neighborhood size combined with a knowledge of the maximum transmission range would allow a node to compute a lower-bound on the local node density.

For the purposes of this chapter, we consider a network to be comprised of those nodes reachable by (multi-hop) messaging from a starting node. As discussed in Section 2.4, nodes that are physically separated (outside of each other's transmission range) cannot communicate. Thus, we can only hope to estimate the number of nodes in a connected topology and we (for now) ignore the problem of physically separate networks. We look at variants of this application designed to estimate both the number of nodes in the connected network and the number of neighbor nodes within the k-hop radius of a given node.

With this application we explore in-network data aggregation and processing, in particular the capability of the network to provide an estimate of its own size. We chose this functionality for several reasons, not in the least because it is an important metric when dealing with very large networks. When using distributed sensor networks for crowd management as explained above, an estimate of network size tells us how many people are in a (part of a) crowd, which may be essential to prevent dangerous situations. Network size estimations will also allow us to efficiently merge a

smaller network with a larger one when networks have become desynchronized, instead of the other way around, as explained in Section 2.4.2. Finally, network size estimations can be used for many network statistics, such as computing average sensor values, node densities and network diameters, allowing one to build an overall picture of a network.

As stated earlier, GMAC was designed with epidemic protocols in mind. In the **NetSize** application discussed in this chapter, a type of epidemic dissemination known as *flooding* is used. In a typical flooding protocol, each node in the network tries to rebroadcast every message it hears to ensure that the message is "flooded" throughout the network such that each participating node is aware of it. **NetSize** represents a particular type of flooding algorithm that shares aspects with *gossiping* algorithms. In this case, the semantics of the data shared by the **NetSize** application allow multiple messages to be merged together in constant space. This means a node can effectively combine and rebroadcast *all* messages it has previously heard in a single message.

The goal of **NetSize** is that every participating node should maintain a local estimate of the size of (number of nodes in) its network. A simple use of the estimates generated by the application would be to adjust the application's message generation rate based on the estimated network size. For example, in very dense neighborhoods the overall network throughput can be increased by limiting the number of broadcasts. As discussed in Section 2.3.1, a local neighborhood containing more nodes than active slots will suffer increased message loss due to collisions. The larger the ratio of neighborhood size to active slots, the more pronounced this problem will become. However, if the application knows that it is in a particularly dense region, it can choose not to broadcast in some gossip rounds in order to reduce traffic on the overloaded medium.

In this chapter, we will begin by explaining the context of our work. We then describe the three algorithms that we will investigate here. In Section 6.3 we discuss the setup of our simulation environment and the particular parameters we will be evaluating. After that, we present our experimental results in Section 6.4. Finally, we conclude and give some indications of potential future research.

6

6.1 Context

In this section we will give an introduction to the subject of *set cardinalities* and several techniques that can be used to *estimate* the cardinalities of very large sets. In this case, our motivation is to estimate the size (i.e., number of nodes) of a wireless ad hoc network. We assume that every node has a unique identifier, and thus we are trying to determine the cardinality of the set of all node ids. This problem is impractical to solve by exact counting for large data sets. This problem is even more pronounced on resource-constrained sensor nodes, where a typical node may have only four kilobytes of RAM for storing an explicit list of identifiers. Thus, we resort to estimators that require far less memory.

6.1.1 Estimation of set cardinality

The cardinality of multiset M is denoted $card(M)$, and represents the number of unique data items contained in M. If the number of unique items in M is small compared to the total number of items in M, a simple iterative count will work well. That is, a node can simply maintain an explicit list of items that it has already seen, checking each new item against the entire list for uniqueness. However, if M is large and many of the items are unique, the list a node would need to maintain would be very long and the memory requirements would make such an algorithm prohibitively expensive to execute.

In [54], the authors propose an efficient algorithm to estimate the cardinalities of multisets. The Flajolet-Martin (FM) sketch they introduce is a simple, but as we will see later a powerful, way to aggregate long datastreams using only a small number of bits. The key feature of this aggregate is that it is unaffected by duplicates and the order in which the data stream is processed. The technique works as follows: Each incoming data item is hashed using a *geometric* hash function, $h(item) = value$, such that the distribution of values is geometric with parameter $\frac{1}{2}$. Let D_n denote the nth data item in the stream.

$$\mathbb{P}(h(D_n) = m) = 2^{-m}$$

for $m, n \geq 1$. Let D be the set of all processed data items. The FM sketch of D is

$$FM(D)[i] = \begin{cases} 1 & \text{if } \exists D_n \in D \text{ such that } h(D_n) = i \\ 0 & \text{else.} \end{cases} \tag{6.1}$$

The aforementioned properties of the sketch (insensitivity to the order and repetition of the entries) can be summarized by the identity

$$FM(D) \vee FM(D') = FM(D \cup D'),$$

where D, D' are sets of data items. Here, and in the rest of the chapter, $\vee$ denotes the coordinate-wise max operation. For an FM sketch $FM(D)$, let $R(D)$ denote the lowest index of $FM(D)$ which contains a 0 value. That is, $R(D) = min\{i : FM(D)[i] = 0\}$. The authors show that with $R(D)$ we can approximate the logarithm of the number of elements of D. Indeed $R(D)$ is close to $\log_2(\varphi|D|)$ where $\varphi \approx 0.775351$. It has standard deviation close to 1.12127. See Theorem 3.A and 3.B of [54] for more details. Hence $\frac{1}{\varphi}2^{R(D)}$ approximates the number of elements of D, and its error is within one binary order of magnitude.

6.1.2 Follow ups

The FM sketch as described above has since found use in a wide array of applications. As such, there have been modifications to and improvements upon the original algorithm. In [55], the authors present a survey of different techniques that can be used for set-cardinality estimation. They analyze the algorithms by looking at both the

processing and storage requirements versus the relative error (i.e., $\frac{card(M)-est(M)}{card(M)}$) of each algorithm.

In [56], the authors consider methods of computing statistics about the data items in a data stream over a sliding window of time. They discuss a method of adapting the traditional FM sketch to operate over a finite time window, with old data items eventually being removed from the sketch. That is, estimating the number of unique data items out of the last N total data items seen from D. This is achieved by maintaining a timestamp, $TS(D)[i]$, for each index in an FM sketch, $FM(D)[i]$. Whenever the value of a particular index is set to 1, the corresponding timestamp is set to be the current time. That is, if D_t is a data item from D that arrives at time t and $h(D_t) = i$, then we set $FM(D)[i] = 1$ and $TS(D)[i] = t$. At every time-step T, we can simply iterate over each index in the FM sketch, and compare the timestamp of that index to the current time. If $T - TS(D)[i] > N$, we set $FM(D)[i] = 0$. In this manner, data items from the stream will "expire" from the FM sketch after N time steps. Computing an estimate of the cardinality of the set can be done in the same manner described for the basic FM sketches in Section 6.1.1. This windowed FM sketch will have a higher standard deviation than a basic FM sketch that has been running for more than N time steps. In the context of sensor networks, this is not an issue if $N \geq d$, where d is the network diameter. During any period of N rounds, a node should see *all* data items in the network. Furthermore, the rate of node arrival and departure will impact the standard deviation of the estimate generated by a windowed FM sketch.

6.1.3 Wireless sensor networks

The authors of [57] apply the FM sketch techniques to the problem of aggregating query results in wireless sensor networks. The use of FM sketches in sensor networks fits well because of the combination of a broadcast-based medium and messages that can be easily aggregated (via a simple Boolean OR operation) without increasing in size. In the paper they demonstrate the effectiveness of their technique through simulations in the Tiny Aggregation (TAG) framework used in TinyDB, described in detail in [58]. As we do here, they also test their solution by using it to count the number of nodes in the network. Our approach of using multiple bitvectors is similar to the technique they present. One important difference is that they assume a single sink node and measure the accuracy of their algorithm from there. In contrast, we ensure that *all* nodes maintain an estimate of the network size.

A recent take on the problem of estimating the size of wireless sensor networks is Extrema Propagation. The authors present the algorithm in [59], demonstrating very accurate results in simulation. They look at several different network topologies and sizes. The main difference with this work is that their technique involves using smaller and less accurate sketches than ours, only 5 bits each. In order to increase the overall accuracy of the algorithm, they bundle 2400 of these sketches in a single message for a message size of 1500 bytes. By averaging the results of all the sketches, the algorithm arrives at an accurate estimate of the network size. By contrast, our algorithm uses much less space, only 24 bytes per message. In addition,

their technique is not easily adapted to the problem of generating arbitrary k-hop neighborhood estimates.

In [60], the authors present a number of different size estimators for sensor networks. Like this chapter, they evaluate their algorithms using grids of simulated nodes. The same authors introduce an improved technique in [61], and evaluate it on even larger simulated networks of up to 10,000 nodes. Their final algorithm demonstrates greater accuracy than both HyperLogLog and Extrema Propagation. However, while their methods achieve more accurate estimates than what we present in this thesis, their algorithms do not take node failure or mobility into account. In addition, the presented algorithms provide whole network estimates, but do not include k-hop neighborhoods.

6.2 Examined techniques

In this section, we explain three different techniques that we will use to estimate the size of a mobile ad hoc network. We begin by describing the mathematical justification for the methods we use. We then move on to elaborate the techniques that we will study in our simulations.

6.2.1 Network size estimation

In the sensor network, we use the FM sketch for random variables which are generated by the nodes: Let $W_{v,t} \in \mathbb{N}$ denote the random value chosen by node v at time t. These random variables are hashed using the same type of geometric hash function discussed in Section 6.1.1, so that $h(W_{v,t}) = X_{v,t}$. Let A be a set of node-time indices, that is a set of pairs (u,s) where u is a node, and s is a time. The FM sketch of A is:

$$FM(A)[i] = \begin{cases} 1 & \text{if } \exists\,(v,t) \in A \text{ such that } X_{v,t} = i \\ 0 & \text{else.} \end{cases}$$

In sensor networks, the wireless connections between nodes are unreliable. Moreover, the network is mobile - nodes can move in space as well as enter and leave the network at will. Hence the neighborhood of a node can change over a short period of time. In such a complex situation we need a clean notation for the k-hop neighborhood, which we provide below.

Let V denote the set of nodes, and $n = |V|$ the size of the network. For $t \in \mathbb{N}$ and $u, v \in V$ we introduce the notation $u \to^t v$. It means that the message broadcast at time t by u was received by v. We use the convention $v \to^t v$, that is, every vertex always receives its own messages. For $k \in \mathbb{N}$, we denote the k-hop neighborhood of v at time t by $N_{v,t,k}$. It is the set of vertices u such that there is a sequence $v =$

$v_0, v_1, \ldots, v_k = u$ such that $v_{i+1} \rightarrow^{t-i} v_i$ for $i = 0, 1, \ldots, (k-1)$. For $k = 0$, we define $N_{v,t,0} = \{v\}$. It is easy to check that

$$N_{v,t,k} = \bigcup_{w \in N_{v,t,1}} N_{w,t-1,k-1} \tag{6.2}$$

for $k \geq 1$. Note that when k is larger than the diameter of the network, the k-hop neighborhood of each node is the whole network, that is $N_{v,t,k} = V$ for all $v \in V$. For later use we also define the sets

$$A_{v,t,k} = \{(w, t-k) | w \in N_{v,t,k}\} \tag{6.3}$$

for $v \in V$ and $t, k \in \mathbb{N}$.

The key idea of [57] is that one can adapt distinct counting algorithms to a sensor network in order to estimate the size of the network. Let us briefly explain their algorithm. At time 0 each node v, independently from each other, chooses a random value $W_{v,0}$.

- Initialization at time 0 : set $B_{v,0}$ to be a vector of 0-s, except for the $h(W_{v,0}) = X_{v,0}$th bit, which is set to 1.
- At each round $t \geq 1$.
  - Broadcast: send $B_{v,t-1}$, and set $B_{v,t} = B_{v,t-1}$.
  - Receive messages: node v gets the message $B_{w,t-1}$ from w. Then v updates $B_{v,t}$ by
    $$B_{v,t} = B_{v,t} \vee B_{w,t-1}.$$
  - Estimate: Let $R_{v,t}$ be the lowest index of $B_{v,t}$ which contains a 0 value. That is, $R_{v,t} = min\{i : B_{v,t}[i] = 0\}$. The estimate is $\frac{1}{\varphi} 2^{R_{v,t}}$.

The definition of $B_{v,0}$ and $A_{v,0,0}$ gives that $B_{v,0} = FM(A_{v,0,0})$. Then a simple induction on k gives that at the end of round t, $B_{v,t} = FM(A_{v,t,t})$.

When we look at the k-hop neighborhood of a node v with k bigger than the diameter of the network, then $N_{v,t,k} = V$. Hence when k is larger than the diameter, we have $A_{v,k,k} = V \times \{0\}$. Since

$$B_{v,t} = FM(A_{v,t,t}) = FM(V \times \{0\}),$$

we get that

$$R_{v,t} = R(A_{v,t,t}) = R(V \times \{0\}).$$

Hence all the nodes arrive to the same network size estimate $\frac{1}{\varphi} 2^{R_{v,t}}$. Recall that the error is big, at least in the order of one binary order of magnitude as mentioned in Section 6.1.1. Environmental factors in the network can increase the error of our estimator. For example, the sudden arrival or departure of a large group of nodes, or a region of the network being subjected to increased interference (radio noise). We show how to reduce the error in the next section.

6.2.2 Multiple bitvectors algorithm

To this point we considered algorithms that estimate the size of the network with only one set of random variables, which were sampled when we initialized the algorithm. However, in a mobile network where the topology and composition of the network changes over time, such an algorithm is not efficient, since after it has converged it provides the same estimate forever.

There are a couple of ways to resolve this problem. The simplest solution is to restart the algorithm after a number of rounds. Even better, we can execute multiple instances of our algorithm in an asynchronous way. For example, we can run 5 instances of the same algorithm, but we start them with 12 rounds of delay, and when an algorithm has run for 60 rounds, we restart it. We call this solution multiple bitvectors, or *MultiBitvector* for short. This solution should also provide more accurate estimates, as the large inherent error will be reduced by averaging the estimates produced by multiple bitvectors.

Unfortunately, none of these solutions provides continuous estimates for the size of the k-hop neighborhood for all k. We give a solution in the next section.

6.2.3 Static bucket algorithm

In order to estimate the size of a node's neighborhood at a (variable) distance of k hops, we need to introduce timestamps, as in [56]. Let L be a parameter. In the FM sketch we replace the bitvectors with vectors of *buckets*, each of which stores an integer between 0 and $L-1$. The bigger the value of a bucket, the fresher the data it corresponds to. When a fresh piece of information arrives, we allow it to overwrite (update) the old one. Moreover, each round the timestamps *age*, that is they decrease by 1. When a timestamp value reaches 0, then we do not decrease it any more. This way we keep information from only the last L rounds - the older information gets discarded. We denote the vector of timestamps of node v at time t as $TFM_{v,t}$. Given this vector of timestamps TFM, for $a \in \mathbb{N}$ let $TFM[a]$ be the bitvector corresponding to the information which is at most a old:

$$TFM[k][i] := \begin{cases} 1 & \text{if } TFM[i] \geq L-k \\ 0 & \text{otherwise.} \end{cases}$$

The precise algorithm, which we call *StaticBucket*, is the following:

- Initialization at time 0: set $TFM_{v,0}$ to be a vector of 0-s, except for the $X_{v,0}$th coordinate, which is set to $L-1$.
- At each round $t \geq 1$.
  - Broadcast: Each vertex $v \in V$ has a vector of timestamps $TFM_{v,t-1}$ from the previous round. It ages the current vector of timestamps:

    $$M_{v,t} := (TFM_{v,t-1} - 1) \vee 0.$$

    (Here $\vee$ is coordinate-wise max.) The node asserts its index, $X_{v,0}$, in the message. That is, it sets $M_{v,t}(X_{v,0}) := L-1$. The node v broadcasts $M_{v,t}$, and sets $TFM_{v,t} := M_{v,t}$.

– Receiving messages: node v gets the message $M_{w,t}$ from w. Node v updates $TFM_{v,t}$ by
$$TFM_{v,t} = TFM_{v,t} \vee M_{w,t}.$$
– Estimate: Let $R_{v,t}[k]$ denote the lowest index of $TFM_{v,t}[k]$ which contains a 0 value. Then $\frac{1}{\varphi} 2^{R_{v,t}[k]}$ estimates the k-hop neighborhood of v at time t.

As above, $TFM_{v,t}$ denotes v's vector of timestamps at the end of round t. That is, after having received all the messages in round t and after all the updates in this round have been done. Let e_l denote the vector whose coordinates are all 0, except for the lth one, which is 1. Then we have

$$M_{v,t} = (TFM_{v,t-1} - 1) \vee \left((L-1)\, e_{X_{v,0}}\right)$$

$$\begin{aligned}
TFM_{v,t} &= M_{v,t} \vee \bigvee_{w \to^t v, v \neq w} M_{w,t} \\
&= \bigvee_{w \to^t v} M_{w,t} \\
&= \bigvee_{w \to^t v} \left((TFM_{w,t-1} - 1) \vee \left((L-1)\, e_{X_{w,0}}\right)\right) \\
&= \bigvee_{w \to^t v} (TFM_{w,t-1} - 1) \vee (L-1) \bigvee_{w \to^t v} e_{X_{w,0}} \\
&= \left(\bigvee_{w \to^t v} TFM_{w,t-1} - 1\right) \vee (L-1)\, FM\left(\left\{X_{w,0} \,\middle|\, w \to^t v\right\}\right) \\
&= \left(\bigvee_{w \to^t v} TFM_{w,t-1} - 1\right) \vee (L-1)\, FM(A_{v,t,1}). \qquad (6.4)
\end{aligned}$$

Now let us take $k \geq 1$, and look at all the timestamps which are at most k old in (6.4). For $k = 1$ we get $TFM_{v,t}[1] = FM(A_{v,t,1})$, that is $TFM_{v,t}[1]$ is the FM sketch corresponding to the 1-hop neighborhood of v at time t. For $k \geq 2$ we get

$$TFM_{v,t}[k] = \bigvee_{w \to^t v} TFM_{w,t-1}[k-1] \vee FM(A_{v,t,1}).$$

Simple induction on k leads to

$$TFM_{v,t}[k] = FM\left(\bigcup_{l=1}^{k} A_{v,t,l}\right)$$

for all $1 \leq k \leq L-1$. Let $R_{v,t}[k]$ denote the lowest index of $TFM_{v,t}[k]$ that contains a 0 value. Then $\frac{1}{\varphi} 2^{R_{v,t}[k]}$ estimates the number of elements of $\bigcup_{l=1}^{k} A_{v,t,l}$. Recall the definition of $A_{v,t,k}$ from (6.3). We see that the sets $A_{v,t,k}$ for $k \geq 1$ are disjoint. Thus

$$A_{v,t,k} = \bigcup_{l=1}^{k} A_{v,t,l}.$$

We get that

$$\frac{1}{\varphi}2^{R_{v,t}[k]} \tag{6.5}$$

estimates the number of elements of $A_{v,t,k}$, which is equal to the size of the k-hop neighborhood of v at time t. Hence we see that using vectors of buckets rather than bits provides estimates for the k-hop neighborhood for all k simultaneously, every round. Notice that when we take k larger than or equal to the diameter of the network, then the formula (6.5) provides an estimate for the size of the whole network. The main problem with the estimate in (6.5) is that it can have a large error. To resolve this problem, we can run multiple instances of this algorithm and combine the results as it was shown in [54], similar to the *MultiBitvector* approach described earlier.

6.2.4 Dynamic bucket algorithm

We propose an algorithm where each node at each round samples a new random variable $W_{v,t}$, and inserts $X_{v,t} = h(W_{v,t})$ into its FM sketch. Clearly, such an algorithm is bound to fail if we do not discard the old values. This is achieved using timestamps, as above. The algorithm proceeds similarly to the *StaticBucket* case. The only difference is that in each round, a node v samples a *new* random number, denoted by $W_{v,t}$, and inserts it in the message: $M_{v,t}(X_{v,t}) := L-1$. Because a node dynamically chooses a new bucket to refresh in each round, we call this variant the *DynamicBucket* algorithm.

As with the *StaticBucket* algorithm discussed above, we have:

$$M_{v,t} = (TFM_{v,t} - 1) \vee \left((L-1)\, e_{X_{v,t}}\right)$$

and:

$$\begin{aligned} TFM_{v,t} &= M_{v,t} \vee \bigvee_{w \to^t v, v \neq w} M_{w,t} \\ &= \bigvee_{w \to^t v} (TFM_{w,t-1} - 1) \vee (L-1) \bigvee_{w \to^t v} e_{X_{w,t}} \\ &= \left(\bigvee_{w \to^t v} TFM_{w,t-1} - 1 \right) \vee (L-1)\, FM(A_{v,t,1}). \end{aligned} \tag{6.6}$$

Now we again take $k \geq 1$, and look at all the timestamps which are at most k old in (6.6). For $k = 1$ we get $TFM_{v,t}[1] = FM(A_{v,t,1})$, that is $TFM_{v,t}[1]$ is the FM sketch corresponding to the 1-hop neighborhood of v at time t. For $k \geq 2$ we get

$$TFM_{v,t}[k] = \bigvee_{w \to^t v} TFM_{w,t-1}[k-1] \vee FM(A_{v,t,1}).$$

Simple induction on k we get that

$$TFM_{v,t}[k] = FM\left(\bigcup_{l=1}^{k} A_{v,t,l} \right)$$

for all $1 \leq k \leq L-1$. That is, in the FM sketch $TFM_{v,t}[k]$, a node w is counted $k-h+1$ times when w is exactly h hops away from v. We again let $R_{v,t}[k]$ denote the lowest index in $TFM_{v,t}[k]$ that does not contain a 0 value. Then $\frac{1}{\varphi}2^{R_{v,t}[k]}$ estimates the number of elements of $\bigcup_{l=1}^{k} A_{v,t,l}$. Recall the definition of $A_{v,t,k}$ from (6.3). We see that the sets $A_{v,t,k}$ for $k \geq 1$ are disjoint. Thus

$$A_{v,t,k} = \bigcup_{l=1}^{k} A_{v,t,l} \setminus \bigcup_{l=1}^{k-1} A_{v,t,l}.$$

We get that

$$\frac{1}{\varphi}2^{R_{v,t}[k]} - \frac{1}{\varphi}2^{R_{v,t}[k-1]} \tag{6.7}$$

estimates the number of elements of $A_{v,t,k}$, which is equal to the size of the k-hop neighborhood of v at time t. Hence we see that the dynamic bucket algorithm also provides estimates for the k-hop neighborhood for all k simultaneously, every round.

Notice that when we take k larger that the diameter of the network, then the formula (6.7) provides an estimate for the size of the network. The main problem with the estimate in (6.7) for large k is that the values of $R_{v,t}[k]$ and $R_{v,t}[k-1]$ are close to each other, and they can even coincide. Again, we can run multiple instances of this algorithm and combine the results to resolve this problem. However, if we want a better estimate for the network size, we can still use only one instance of this algorithm, but average the estimates in (6.7). It provides the estimate

$$\frac{2^{R_{v,t}[k+q]} - 2^{R_{v,t}[k]}}{\varphi q} \tag{6.8}$$

It estimates the average number of nodes present in the network in between round $t-k-q+1$ and $t-k$. In order to decrease the chance that $R_{v,t}[k+q] = R_{v,t}[k]$ we have to set q such that it is comparable to k.

Remark 1. Most of the sketches listed in [55] can be adapted for the dynamic setting, with a similar use of timestamps.

6.3 Experimental setup

6

In this section, we explain the setup, execution and analysis of our simulations. Once again, the costs (both in time and money) of performing real experiments with thousands of nodes are prohibitively high, so we resort to experimentation via simulation. The details of our simulation environment were covered in Section 4.1, and here we describe only the specific parameters used in this chapter.

6.3.1 Simulator parameters

Clocks Here we investigate GMAC's application-level behavior, and we are not interested in synchronization directly. As such, we choose to use the default *MaxClockDrift* value of ±20 ppm.

Table 6.1: Network topologies investigated in this chapter

Nodes	Layout	Diameter	Dimensions	Spacing
1024	32×32	32	$640m \times 640m$	20m Grid
1024	64×16	64	$1280m \times 320m$	
1024	128×8	128	$2560m \times 160m$	

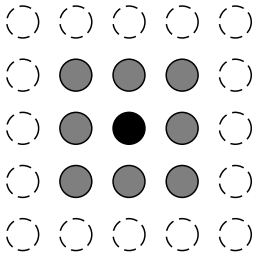
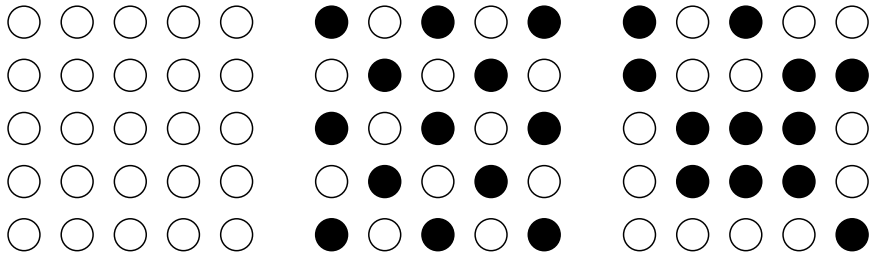


Figure 6.1: Graphical representation of the transmit range using simulated $0.5mW$

Network topology One of the chief parameters to investigate is the diameter of network. The more hops required for a piece of data to traverse from one end of the network to the other, the longer it will take our estimators to react to changes happening in other parts of the network. In order to better assess the strengths and weaknesses of our estimation algorithms, we investigate the effect of changing the diameter of our simulated networks. In particular, we look at a fixed network size and vary the layout of the nodes. In all of our experiments the nodes are deployed in a regular *grid* pattern. Nodes are deployed in an $N \times M$ grid, with rows (and columns) placed $20m$ apart (see Table 6.1). It is important that the networks we examine are connected, because otherwise complete dissemination of information would be impossible. Note that this is in contrast to our synchronization topologies, where mobility patterns that (temporarily) split the network into separate subnets are an important aspect of our investigation. Note that we simulate some of the *effects* of node mobility by including scenarios with node failure, discussed in detail later in this section. In many aspects, a node joining or leaving a network is indistinguishable from a node simply moving into or out of range of the other nodes in the network.

Transmission density The transmission power of the simulated radios, along with the network topology, determines the connectedness of the entire network. By increasing a simulated node's transmission power, the simulator increases the node's transmit range. This effectively decreases the diameter of the network. Conversely, the diameter of the network can be increased by decreasing the nodes' transmission power. Based upon the default grid spacing of $20m$, we have chosen to use $MaxTxPower = 0.5mW$ as our default setting. In Figure 6.1 we show a group of nodes spaced $20m$ apart, depicting the transmission range from the perspective of the sender (black) and the potential receivers (gray).



(a) No failures (b) Symmetric failures (c) Random failures

Figure 6.2: Graphical representation of three of the node activity scenarios

6.3.2 Application variants

We look at three different application-level algorithms to estimate network size, and two different kinds of application messages.

- <**StaticBucket**> This is the timestamped extension to the basic FM sketch which includes using *buckets* rather than bits. The application data in a message consists of 24 1-byte buckets, for a total of 24 bytes.
- <**DynamicBucket**> This is the timestamped extension using buckets and randomly generated bucket ids in each round. The application data in a message consists of 24 1-byte *buckets*, for a total of 24 bytes.
- <**MultiBitvector**> This is the method of utilizing multiple bitvectors to smoothen estimates. The application data in a message consists of $M = 8$ sketches of $N = 24$ bits each, for a total of $\frac{M \times N}{8} = 24$ bytes.

6.3.3 Scenarios

We utilize different activity scenarios in order to evaluate different aspects of the size estimation.

Active

All nodes are active for the entire duration of the simulation

Symmetric failure

Half of the nodes fail (turn off) 25% of the way through the simulation, and are reactivated 75% of the way through the simulation. Nodes are failed in a "checkerboard" fashion, where all of a node's direct north/south/east/west neighbors will be in the opposite state and its diagonal neighbors will be in the same state.

Random failure

Half of the nodes fail (turn off) 25% of the way through the simulation, and are reactivated 75% of the way through the simulation. In contrast to the symmetric failure scenario, in this case the nodes to fail are selected at random with the caveat that the remaining active nodes will continue as a connected network. That is, a node that would break the remaining active nodes into multiple unconnected networks cannot be selected to fail.

Churn

In these scenarios, a particular percentage of nodes will fail 25% of the way through the simulation. Each of these nodes will be offline for a random amount of time, $30s <= t <= 120s$. Whenever a failed node reactivates, another node will be randomly selected to fail for a random duration. The same rule regarding network partitions as above applies, and we introduce an additional constraint: a node that has just recovered from failure cannot be selected to fail for at least $30s$.

6.4 Evaluation

Here we evaluate the performance and accuracy of the three estimation techniques described in Section 6.2. We begin by explaining the measurements we use to evaluate the three algorithms. We then present the results of a series of experiments designed to compare the different algorithms with respect to the estimation of the size of the entire network. Finally, we evaluate the accuracy of the most promising technique of the three when estimating a node's *k-hop neighborhood*. We look at a range of values for k, but we are most interested in the low values ($k = 1, 2, 3$, representing a node's *local* neighborhood) and the high values ($k \approx diameter$).

6.4.1 Measurements

As we are chiefly interested in accurate estimation of the network size, our main observable measurement is the per-round state of each node's estimator. Every simulated node records the entire contents of its bitvector at the end of each simulated round. A node's network size estimate for a given round is deterministically based on the state of its bitvector (or bucket vector) in that round. Thus, after the simulation has completed, we can evaluate different estimation parameters or algorithms on the logged data to see which performs most accurately. For example, we can experimentally determine an appropriate value for q in Equation (6.8).

We primarily use the percentage error in estimated network size in the plots in the following section. This is computed as $Err_{v,r} = (Est_{v,r} - |V_r|) \times \frac{100}{|V_r|}$, where $Est_{v,r}$ is

the estimate computed by node v in round r and V_r is the set of active nodes during round r. Since we are presenting the results of networks of 1024 nodes, plotting each node's estimate is impractical. As such, we can compute the mean error across all nodes for a given round:

$$Err_r = \frac{1}{|V_r|} \sum_{v \in V_r} Err_{v,r}$$

Instead of a simple arithmetic mean, we can compute the Root Mean Square Error:

$$RMSE_r = \frac{1}{|V_r|} \sqrt{\frac{1}{|V_r|} \sum_{v \in V_r} (Est_{v,r} - |V_r|)^2}$$

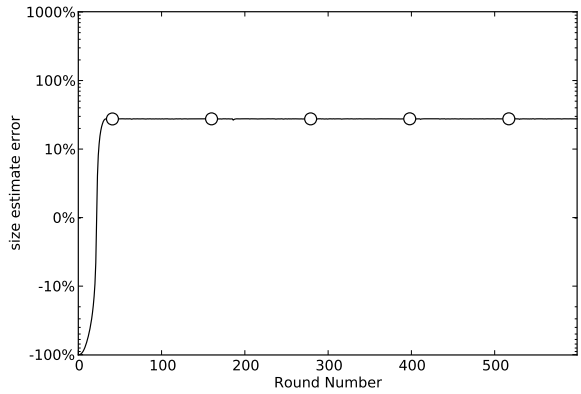
These metrics are mainly interesting in order to compare different estimation algorithms.

6.4.2 Comparison of estimators

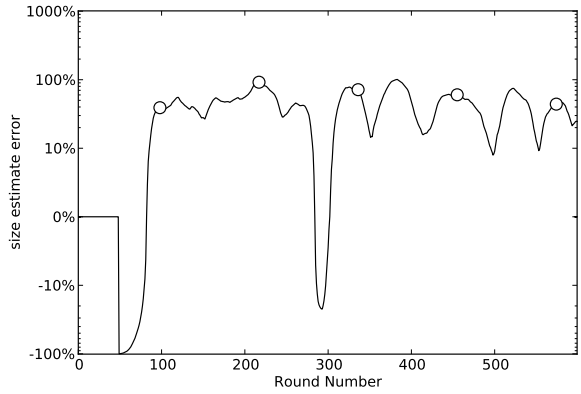
Here we look at the accuracy of our three techniques. We can compare the three algorithms fairly by ensuring that all three have the same message complexity. That is, we use a message size of 24 bytes in each case as explained in Section 6.3.2. We begin our evaluation with fully active networks, where the multiple bitvectors should have an advantage. Then we look at three scenarios involving node failure, where the timestamped bucket techniques should outperform the flat bitvectors.

We start by looking at the performance of our three estimation algorithms in the active scenario. Figure 6.3 shows the results of each estimator on the 32×32 grid topology. This topology has a diameter of 32, so we choose $k \geq 32$ (for StaticBucket and DynamicBucket algorithms) in order to estimate the size of the whole network. Specifically, we choose $k = 32$ for the StaticBucket algorithm, and $k = 64$, $q = 32$ for the DynamicBucket algorithm. Recall that the q parameter was introduced in Section 6.2.4 in order to compensate for over-counting by the DynamicBucket technique.

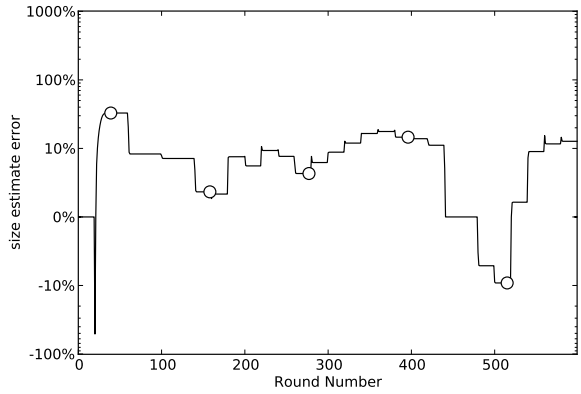
In Figure 6.3a, we see that the StaticBucket algorithm quickly reaches a stable estimate of the network size, with an estimation error of about +20%. Because there is no mobility or failure in the simulated network, once the algorithm converges, the result does not change for the remainder of the runs. The results of the DynamicBucket method are significantly more variable. As can be seen in Figure 6.3b, this estimator never converges because nodes continually select new bucket IDs throughout the runs. Our attempts at compensating for this (expected) overestimation are only partially effective. The estimation error varies from about −10% to +110% during the simulated runs. This variability will make it difficult for nodes to be confident in size estimates generated by the DynamicBucket algorithm. Finally, in Figure 6.3c,



(a) StaticBucket, *active*

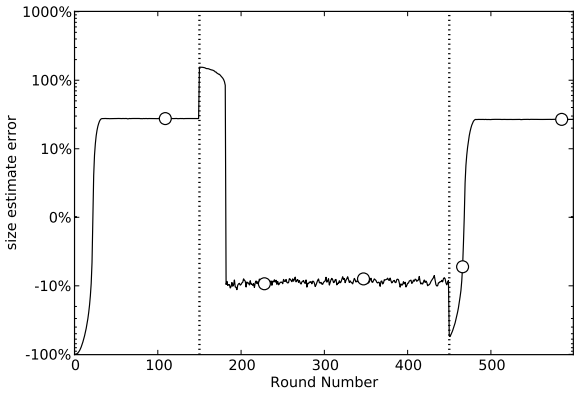


(b) DynamicBucket, *active*

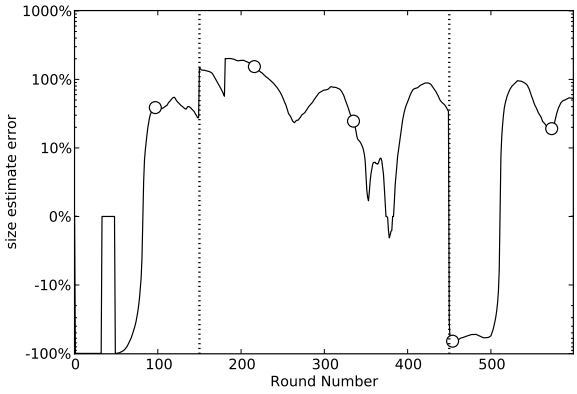


(c) MultiBitvector, *active*

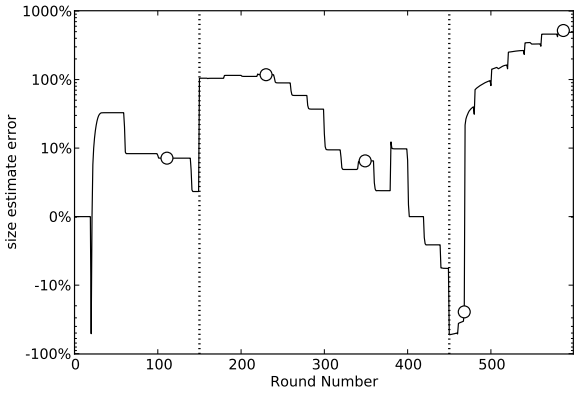
Figure 6.3: The performance of our three algorithms for the *active* scenario, 32×32 grid



(a) StaticBucket, *50% random failure*



(b) DynamicBucket, *50% random failure*



(c) MultiBitvector, *50% random failure*

Figure 6.4: The performance of our three algorithms for the *random failure* scenario, 32×32 grid

we present the results for the MultiBitvector estimator. This estimator offers something of a compromise between the quick to converge behavior of StaticBucket and the continuous variability of DynamicBucket. The accuracy of MultiBitvector also falls between that of StaticBucket and DynamicBucket, ranging from −10% to +30%.

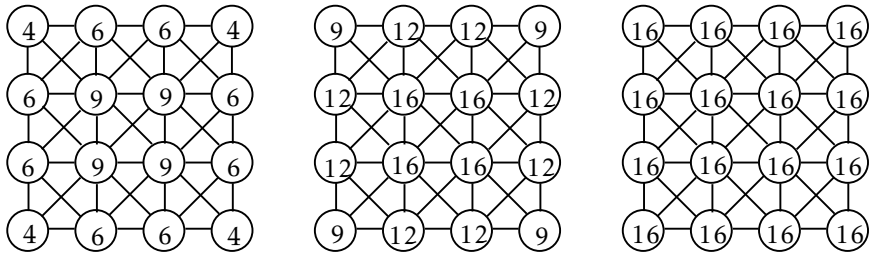
Though both StaticBucket and MultiBitvector perform adequately in fully active networks, we are far more interested in the results for scenarios involving node failure. As discussed previously, an estimator that performs well in variety of network topologies and failure scenarios is essential for our domain. Social ad hoc networks will necessarily involve a high degree of node mobility and node failure must be handled gracefully. As such, we now proceed to evaluate the three estimators in our *random failure* scenario.

In the three sub-figures of Figure 6.4, the vertical dotted lines at round number 150 and 450 represent the beginning and end of the period of node failure, respectively. Between the two vertical lines, the network experiences node failure, while to the left or right of this region all nodes are active. The reader should note that all algorithms show an upward spike in round 150 and a corresponding downward spike in round 450. This is due to the sudden disappearance (or reappearance) of 512 nodes. As the simulations proceed, the estimators adapt to the new network size relatively quickly.

We begin with Figure 6.4a, where we present the results for the StaticBucket algorithm in the *random failure* scenario. The far right and far left of the graph appear identical to the results from the *active* scenario in Figure 6.3a. During the period of node failures, the StaticBucket algorithm produces a fairly stable estimate, with an error of approximately −10%. Next we return to the DynamicBucket algorithm, in Figure 6.4b. We again see that this algorithm generates extremely variable results, though they continue to be bounded between approximately −10% to +110% during the simulated runs. Note that the DynamicBucket algorithm exhibits a delay of approximately 64 rounds in responding to network changes. This is a result of using a higher k value ($k = 64$) than in the StaticBucket case, signifying data items do not expire for 64 rounds. Lastly, we examine the MultiBitvector method in Figure 6.4c. We can see that the results from the beginning of the simulation runs are comparable to those from the *active* scenario. When node failures begin in in round 150, we see the expected jump in estimated network size, which then gradually decreases during the period of failure as old data are eventually evicted from the bitvectors. Somewhat surprising is the continually growing estimate evident after the period of node failure ends. This behavior is because when the failed nodes restart operation in round 450, they are not initially synchronized with the nodes that were continuously active. Though these restarted nodes quickly resynchronize, first they end up "polluting" the distributed bitvector array by setting bits in the wrong bitvector. These additional bits get distributed throughout the network and cause the nodes that didn't fail to overestimate the network size.

6.4.3 K-hop estimation

Here we evaluate the accuracy of estimating k-hop neighborhoods, for various values of k. Our interest here is whether we can effectively use one algorithm for nodes to



(a) k=1 (b) k=2 (c) k=3

Figure 6.5: Graphical representation of k-hop neighborhoods

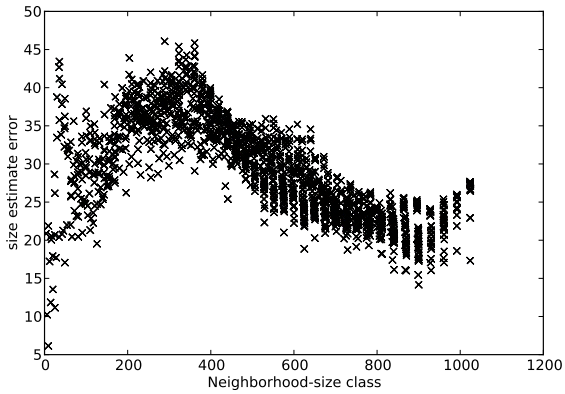
both estimate the size of the whole network and any of their k-hop neighborhoods. In this section, we will only look at the StaticBucket algorithm, as the DynamicBucket was shown to be far too variable for accurate size estimation.

Our method of evaluation is to compute the mean k-hop estimate for each *neighborhood class*. We can group nodes into neighborhood classes based on the number of neighbors a node has at a maximum distance of k hops, for $k = 1..32$ and $k = 128$. Each node v has exactly $Nbr_{v,k}$ neighbors within k hops. For all nodes, $Nbr_{v,k} = |V|$ for k larger than or equal to the network diameter. We generate a single data point for each unique combination of $(k, Nbr_{v,k})$. Note that a particular neighborhood class size may exist at a variety of different k values. For example, in Figure 6.5 we see that neighborhood classes of size 9 exist at $k = 1$ and $k = 2$, while neighborhood classes of size 16 exist at $k = 2$ and $k = 3$. By computing the mean estimate of all nodes in a particular neighborhood class, we can get an idea of how well our estimator performs at various neighborhood sizes. In order to reduce our data to just a single point representing each class, we take the mean estimate at round 300, exactly half-way through the simulation. This round was chosen because it gives the estimator a chance to converge in scenarios with or without node failure, and gives us insight into how the estimator handles failure scenarios since round 300 occurs *during* the failure part of the simulated runs.

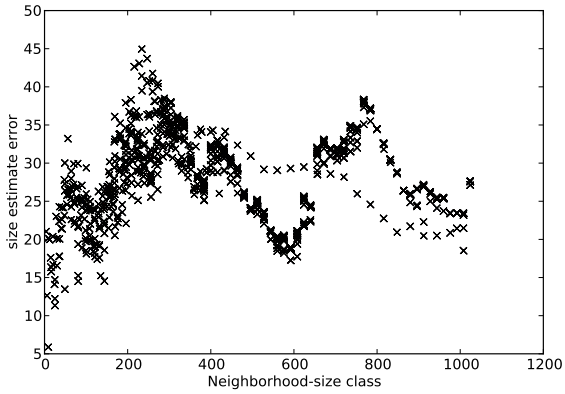
In Figure 6.6 we present such an analysis. This figure depicts the results of 50 simulated runs of each topology that we study. Figure 6.6a shows the 32×32 grid with diameter 32, Figure 6.6b shows the 64×16 grid with diameter 64, and Figure 6.6c shows the 128×8 grid with diameter 128. We can see that regardless of the diameter of the network or the neighborhood class size, the error in the StaticBucket algorithm's estimate is strongly bound, $0\% \leq Err \leq 50\%$. The actual error varies quite a bit within that range, meaning that nodes certainly cannot assume their estimate is perfectly accurate, but the accuracy does improve as the size of the network being estimated grows.

Finally, in Figure 6.7 we look at the performance of our estimator on the 64-hop diameter network in the face of node failures. Figure 6.7a shows the results of the random failure scenario, and Figure 6.7b shows the churn scenario. In both cases we can see that the estimates are even more variable than in the no failure scenario, as can be expected. However, even here the estimation error is (with few exceptions) bound between $-50\% \leq Err \leq 50\%$. The outlying data point in the results of the

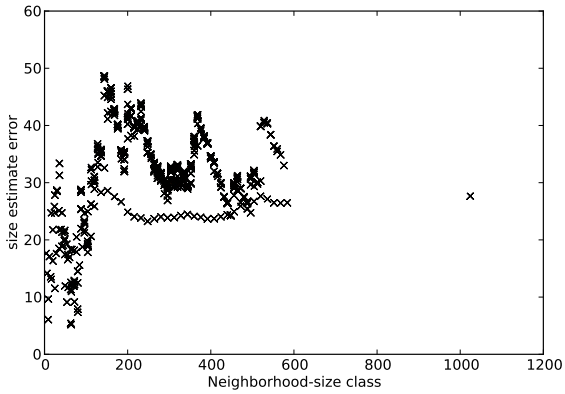
churn scenario is from the value $k = 128$. Since the diameter of the network is less than 128, such an estimate should include the entire network. The problem lies in the fact that it takes 128 rounds for a "filled" bucket to "empty" with $k = 128$. That is, after the node filling a particular bucket fails, there is a 128 round delay before the estimator will see that bucket as empty again (assuming no other node *also* fills that same bucket). Thus, as nodes constantly fail and recover during the churn scenario, the high-order buckets stay filled. This issue is much less pronounced at lower k values, and these estimators respond to changes in the neighborhood size with lower latency.



(a) 32×32

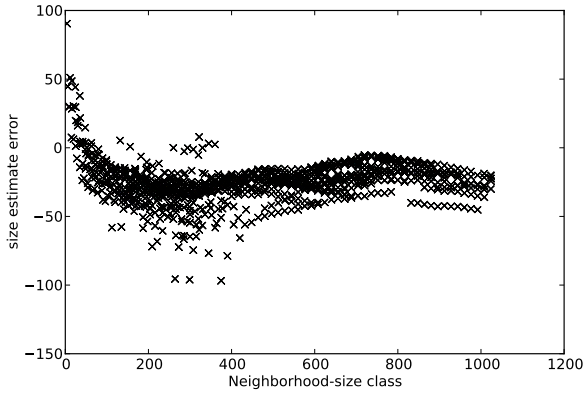


(b) 64×16

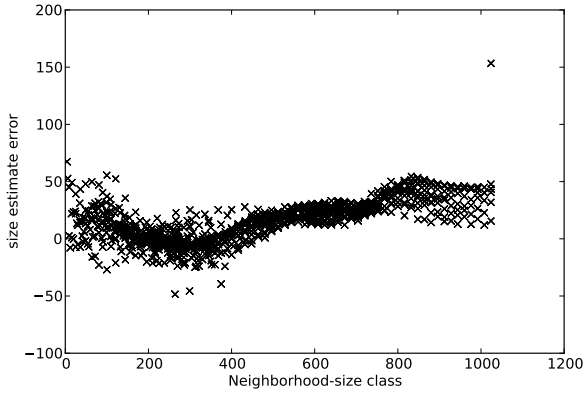


(c) 128×8

Figure 6.6: The performance of *StaticBucket* for the *active* scenario, 1024 nodes



(a) Random failure



(b) Network churn

Figure 6.7: The performance of *StaticBucket* with network failure

6.5 Conclusions

In this chapter we presented a simple and scalable epidemic application designed to generate continuous estimates of the current network size. We discussed the history and theory of estimating set cardinalities, as well as how such techniques can be used to estimate the number of wireless sensor nodes actively participating in a completely decentralized network. We described and evaluated three different algorithms, each offering unique advantages and disadvantages. We compared these algorithms using identical message complexity in each case, since this is a crucial factor in the energy cost of running an application. In addition to looking at the effects of the network diameter on the accuracy of our estimators, we also investigated the effects of node failures. We simulated scenarios with no failure, persistent failure, and transient failure (churn).

Our simulations demonstrate that both the MultiBitvector and StaticBucket estimators work well in networks without node failure. In such scenarios, the StaticBucket algorithm quickly converges to a final estimate, while the MultiBitvector algorithm varies continually throughout the simulated duration. Nevertheless, both offer adequate accuracy (±50%) for many purposes. In particular, we propose that such an estimate could be a useful metric for making cluster merge decisions. That is, if each node maintains an estimate of its network's size, this estimate can be sent with **join** messages. Then, nodes can decide to always merge a smaller cluster into a larger one, minimizing the number of nodes that must re-synchronize.

We additionally demonstrated that the StaticBucket estimator was capable of generating continuous k-hop neighborhood estimates, a powerful advantage over the MultiBitvector algorithm. Such estimates of only the node's *local neighborhood* can allow the nodes to react to significant changes in network topology or other environmental conditions. We showed that this k-hop neighborhood estimation generates larger errors in very small neighborhood sizes, but performs acceptably across a large range of values for k.

Although this chapter has shown the StaticBucket to be a potentially valuable tool for enabling social ad hoc networks, there are many opportunities for extending this work. An important one is using these estimates in order to adapt a node's duty cycle to the local node density. That is, if there are many neighbors in a node's neighborhood, it should broadcast less often or extend its active period in order to relieve congestion. Conversely, a node can shorten its active period if there are few other nodes in its neighborhood, as the extra slots will only be wasted. In addition to improving the cluster merge behavior mentioned above, providing self-adaptation with regard to density was a motivating factor in this investigation.

Another important follow up is to investigate the performance of these algorithms in the presence of node mobility. As explained earlier, our activity scenarios provide an easier to analyze proxy for mobility, but are not the same. In Chapter 5, we demonstrated that GMAC can function properly in extremely large scale mobile networks, but it remains to be seen how this application would perform.

Finally, it is crucial to test these algorithms in real-world networks. Simulation is an essential part of experimenting with wireless sensor networks, but can never replace real deployments. Such deployments often reveal unanticipated problems with algorithms or assumptions that are not seen in the simulator. Unfortunately this avenue of investigation took place late in our research, and we did not have an opportunity to evaluate the application in an appropriate social ad hoc network. We expect to perform experiments of the scale required to validate our simulations in the future, and look forward to confirming the results presented here.

6

Part IV Discussion

7. Discussion

7.1 Summary

In this thesis we presented a thorough analysis, from hardware details to application-level simulation, of the operation of a family of low duty-cycle networking protocols collectively called GMAC. In the course of this work, we investigated a number of improvements to the basic GMAC protocol, designed to enable our vision of social ad hoc networks. In particular, we evaluated the scalability of the synchronization provided by these protocols in networks ranging from a few dozen nodes to over four thousand mobile nodes. Additionally, we analyzed the behavior of a simple but powerful epidemic application implemented using GMAC's API. Finally, we presented detailed descriptions of four of our real deployments with critical analysis of what went wrong and what went right with our complete solution.

In Chapter 2 we presented the MyriaNed platform, for which the GMAC family of protocols was originally designed. We explained details regarding the radio and timing hardware used by the platform, as well as the CPU, memory and other significant information. We also provided a description of the GMAC family of protocols, including a comparison with more traditional network protocols. We presented motivation underlying GMAC and an analysis of its network synchronization model. In addition, we detailed the API that GMAC presents to the application layer and explained the modules GMAC uses for slot allocation and neighbor synchronization. An important contribution of this chapter is breaking up the problem of frame-based network synchronization into a number of fundamental subproblems. This approach allowed us to clearly analyze the problems we faced and draw meaningful conclusions from our simulations and experiments.

In Chapter 3 we presented a critical analysis of our four most interesting real-world experiments, including an accurate assessment of GMAC's synchronization behavior in an actual social ad hoc network consisting of over one hundred nodes. We found that GMAC, with the addition of synchronization improvements described in this thesis, is a viable solution for real-world social ad hoc networks. This validation of our simulated results shows that the progress made towards our more abstract research goals is realizable in practice. Finally, we learned that successful social ad hoc networking experiments require careful planning and *lots* of testing. The experience gained makes increasing the scale of future real-world deployments more tractable.

In Chapter 4 we evaluated GMAC in a simulated environment using static network topologies. We found that GMAC's maintenance of existing network synchronization performed well across a wide variety of simulated clock errors and activity patterns, but the group merge behavior was lacking. We proposed a number of improvements to the detection, decision and notification sub-problems pertaining to merging synchronized groups and analyzed their performance. We found that using

active detection with deterministic decisions via *cluster tags* and a simple notification strategy, GMAC was able to quickly synchronize a network of over four thousand initially completely unsynchronized nodes to a single duty schedule. In addition, we found that GMAC not only operates correctly at the standard duty cycle of 0.68%, but potentially far lower. Combined with the potential reduction in guard times due to better-than-expected synchronization, we estimate GMAC will operate correctly at a duty cycle of less than 0.1%. Though not presented in this thesis, experiments in our simulator indicate this estimation should be accurate. This is an important result towards to our first research goal of low power operation, though further exploration is required to determine any possible lower bound on the duty cycle.

In Chapter 5 we continued our analysis of GMAC's synchronization behavior in the context of mobile networks. We proposed several additional improvements to the GMAC's behavior. Using a variety of pre-generated mobility patterns, we showed that our improvements enable GMAC to rapidly and *consistently* synchronize networks of four thousand simulated nodes. In fact, our most recent MyriaNed deployments validate, at least on a small scale, that the improvements we added to GMAC's synchronization behavior are effective. This result demonstrates significant progress on our research goals of mobile networks of $O(10,000)$.

Finally, in Chapter 6 we presented an epidemic application designed to estimate the size of the network in which the executing node participates. We showed that, in collaboration with its neighbors, a node can maintain an accurate estimate of not only the size of the whole network, but also its local k-hop neighborhood for any desired value of k. Such estimates of basic network parameters are an important foundation for our final research goal of adaptability.

7.2 Conclusions

To conclude, the work presented in this thesis demonstrates that our solution, an improved version of GMAC, is able to maintain tight inter-node synchronization in a completely decentralized manner. This synchronization allows GMAC to operate at extremely low duty cycles while avoiding any dependency on *special* nodes, such as gateways or cluster heads. Furthermore, our solution functions effectively in both static and mobile network topologies, with sizes ranging from 64 to 4096 nodes. These features are an absolute requirement in order to enable the social ad hoc networks which were the goal of this thesis. Beyond this, we showed that GMAC provides a viable platform for building epidemic-based applications for social ad hoc networks, capable of introspectively discovering local network parameters and allowing for the deployment of novel tools for investigating real-world social behavior. The potential uses of these are vast, and we expect this networking domain to continue mature in the coming years.

Our successful real-world experiments allow us to conclude that we *can* build cohesive social ad hoc networks out of a massive number of wearable wireless sensor devices. Our protocols are very energy efficient, and also provide a predictable node

lifetime. By utilizing duty cycles well below 1%, GMAC is able to achieve significant energy savings when compared to other MAC layers. The final version of GMAC is able to quickly discover and synchronize nodes new to the network, and also allows nodes to simply fail or leave the network without significantly affecting its operation. Furthermore, though we explored the problems of synchronization and group merging in the context of GMAC, many of our results are applicable to other MAC layers. For example, our evaluation of active vs. passive detection and methods of deciding which group should merge into which apply to any MAC layer that utilizes duty-cycling. These techniques do not depend on any hardware specific to the MyriaNed platform, and assume only that nodes use a short active period followed by a long inactive period. Such a model is common to many MAC layers that utilize duty-cycling.

Although this thesis has only scratched the surface of what is possible with social ad hoc networks, we have achieved many of our original goals for this project. With regard to our first research goal of low-power operation, we did not determine a lower bound, but we did explore duty cycles on the order of 0.1%. We also established bounds for what level of synchronization is required to enable communication in these mobile networks, and ensured that GMAC maintained those bounds in a wide variety of experimental settings. We made good progress on our second research goal of highly scalable operation, demonstrating correct function across networks of different sizes and topology. Our third goal of insensitivity to node mobility has been achieved, particularly visible in our results from Chapter 5 which show even better synchronization bounds in mobile networks than for static ones. Finally, we did not fully achieve our final goal of adaptivity to local network conditions, but did lay the foundation of a proper solution with our work in Chapter 6.

7.3 Future work

Though we have made significant progress towards our original goals and look at this research as a success, there remains much work to be done. In this section we discuss several lines of research to continue investigating.

7.3.1 Scale

First and foremost, we did not reach our goal of operation in a network of at least ten thousand nodes. Especially in the context of the real-world experiments, we need to increase the scale of our networks. One idea would be to distribute a version of our wearable nodes to the attendees of large music festival or sporting event. For example, the Lowlands music festival (`http://lowlands.nl`) attracts approximately 55,000 guests and is held each year in the Netherlands during the course of a weekend in August. Using a number of sniffer nodes distributed throughout the venue, the hosts could anonymously monitor the concert-goers in real time. This could be used for things as pedestrian as efficiently distributing staff amongst the various refreshment stands, or as important as trying to avoid injuries or deaths resulting from trampling around overcrowded entrance/exit areas.

7.3.2 Slot allocation

We have largely ignored an important aspect of GMAC, that of slot allocation. Throughout this thesis, nodes simply choose their transmit slot at random from the slots in the active period. No attempt is made at coordinating this slot selection with neighboring nodes, ensuring the collisions will certainly occur on a probabilistic basis. However, if a node's neighborhood is relatively stable and enough active slots are available, it could be able to negotiate a stable slot allocation that allows all neighbors to communicate collision-free. There is significant existing research in this area, and evaluation would simply mean implementing new *strategy* modules for GMAC.

7.3.3 Adaptivity

We demonstrated that GMAC applications can correctly estimate local network parameters like density, but we did not show how GMAC can utilize this information to adapt its operation to the current conditions. For example, in dense neighborhoods, nodes could decide to use a longer active period than in other parts of the network. These additional active slots would serve to reduce congestion in the dense region by providing more transmission slots to nodes in the area. This could potentially be implemented as an addition to the slot allocation work discussed above. As another example, potential energy savings could be realized by adaptively changing the rate which **join** messages are sent. Rather than sending a **join** message every frame, nodes could keep track of the last time they heard a **join** message and adjust the frequency of their **join** broadcasts accordingly. If a node has not received a **join** message in a long time, it is likely that there are no unsynchronized nodes in its neighborhood. Thus, the active detection frequency could be reduced (e.g., to one **join** broadcast every second frame), lowering energy consumption.

7.3.4 Improved hardware

While we can implement anything we would like in simulation, in the real-world we are restricted by the current MyriaNed hardware. The existing nodes are several years old, and new components are cheaper, lighter and more energy-efficient. In particular, a more fine-grained timer perhaps on the order of 1 MHz, would allow for more precise on-node timing and synchronization adjustments. Additionally, a faster and more flexible radio would allow GMAC to send larger packets and increase energy efficiency. By including more application data in each broadcast, the energy cost per bit of application data sent will drop. Furthermore, a radio with faster transitions between send and receive mode can help save energy by delaying radio activation even longer.

7.3.5 Synchronization maintenance

Although we demonstrated that GMAC's *median* synchronization algorithm works well, the models created by the authors of [62] showed that in certain situations it can cause the participating nodes to spontaneously de-synchronize. While we have not observed this behavior in our simulations or experiments, there are certainly opportunities for investigating alternatives. One such alternative popular in the literature is based on pulse-coupled oscillators [63] [29], known as *firefly* synchronization because it is modeled after swarms of fireflies. Any improvement to the network synchronization can generally be translated directly into energy savings by reducing slot guard times and delaying radio activation.

7.4 Final thoughts

In conclusion, we have shown that it is achievable to develop *ultra-low* duty cycle wireless ad hoc networks consisting of thousands of mobile nodes. Our solution provides a predictable device lifetime by ensuring all devices consume their energy store at the same *constant* rate. We have shown that our solution is resilient to node failure and that it is effective even in networks exhibiting arbitrary node mobility. This is a unique result, as CSMA protocols are often preferred in such environments. In addition, we demonstrated that our solution is good candidate for a scalable platform suitable for developing applications providing *in-network* data aggregation and self-assessment of fundamental network parameters. Finally, we provided a number of examples of interesting research still to be performed in this exciting area.

References

References

1. Gaba A, Voulgaris S, Steen M. Group Monitoring in Mobile Ad-Hoc Networks. *Proceedings of the Third International Conference on E-Democracy*, 2009.
2. Roberts L. G. Aloha packet system with and without slots and capture. *ACM SIGCOMM Computer Communication Review*, 5(2):28–42, 1975.
3. Polastre J, Szewczyk R, Mainwaring A, et al. Analysis of wireless sensor networks for habitat monitoring. *Wireless sensor networks*, pages 399–423, 2004.
4. Goense D, Thelen J, Langendoen K. Wireless sensor networks for precise phytophthora decision support. In *5th European Conference on Precision Agriculture (5ECPA), Uppsala, Sweden*. Citeseer, 2005.
5. Assegei F. Decentralized frame synchronization of a TDMA-based wireless sensor network. *Master's Thesis, Eindhoven University of Technology, Department of Electrical Engineering*, 2008.
6. Choudhury T, Pentland A. Sensing and modeling human networks using the sociometer. In *Proc. the 7th IEEE International Symposium on Wearable Computers (ISWC2003)*, pages 216–222, 2003.
7. Langendoen K. The mac alphabet soup, 2009. `http://www.st.ewi.tudelft.nl/~koen/MACsoup/`.
8. Singh S, Raghavendra C. S. Pamas - power aware multi-access protocol with signalling for ad hoc networks. *ACM SIGCOMM Computer Communication Review*, 28(3):5–26, 1998.
9. Chen J.-C, Sivalingam K. M, Agrawal P. Performance comparison of battery power consumption in wireless multiple access protocols. *Wireless Networks*, 5(6):445–460, 1999.
10. Sivalingam K. M, Chen J.-C, Agrawal P, et al. Design and analysis of low-power access protocols for wireless and mobile atm networks. *Wireless Networks*, 6(1):73–87, 2000.
11. Hedetniemi S. M, Hedetniemi S. T, Liestman A. L. A survey of gossiping and broadcasting in communication networks. *Networks(New York, NY)*, 18(4):319–349, 1988.
12. Demers A, Greene D, Hauser C, et al. Epidemic algorithms for replicated database maintenance. In *Proceedings of the sixth annual ACM Symposium on Principles of distributed computing*, pages 1–12. ACM Press New York, NY, USA, 1987.
13. Bavelas A. Communication patterns in task-oriented groups. *The Journal of the Acoustical Society of America*, 22(6):725–730, 1950.
14. Hajnal A, Milner E, Szemerédi E. A cure for the telephone disease. *Canad. Math. Bull*, 15(3): 447–450, 1972.
15. Elson J, Römer K. Wireless sensor networks: A new regime for time synchronization. *ACM SIGCOMM Computer Communication Review*, 33(1):154, 2003.
16. Sivrikaya F, Yener B. Time synchronization in sensor networks: A survey. *IEEE network*, 18 (4):45–50, 2004.
17. Rahamatkar S, Agarwal A, Kumar N. Analysis and Comparative Study of Clock Synchronization Schemes in Wireless Sensor Networks. *Analysis*, 2(03):536–541, 2010.
18. Langendoen K. Medium Access Control in Wireless Sensor Networks. In Wu H, Pan Y, editors, *Medium Access Control in Wireless Networks, Volume II: Practice and Standards*. Nova Science Publishers, 2007.
19. Polastre J, Hill J, Culler D. Versatile low power media access for wireless sensor networks. In *Proceedings of the 2nd international conference on Embedded networked sensor systems*, pages 95–107. ACM, 2004.
20. El-Hoiydi A, Decotignie J.-D. Wisemac: An ultra low power mac protocol for multi-hop wireless sensor networks. In *Algorithmic Aspects of Wireless Sensor Networks*, pages 18–31. Springer, 2004.
21. Ye W, Heidemann J, Estrin D. An energy-efficient MAC protocol for wireless sensor networks. In *21st Annual Joint Conference of the IEEE Computer and Communications Societies (INFOCOM)*, pages 1567–1576. IEEE, 2002.
22. Van Dam T, Langendoen K. An adaptive energy-efficient MAC protocol for wireless sensor networks. In *Proceedings of the 1st International Conference on Embedded Networked Sensor Systems*, pages 171–180. ACM, 2003.
23. Ye W, Silva F, Heidemann J. Ultra-low duty cycle MAC with scheduled channel polling. In *Proceedings of the 4th International Conference on Embedded Networked Sensor Systems*, pages 321–334. ACM, 2006.
24. Rajendran V, Obraczka K, Garcia-Luna-Aceves J. Energy-efficient, collision-free medium access control for wireless sensor networks. *Wireless Networks*, 12(1):78, 2006.
25. Cidon I, Sidi M. Distributed assignment algorithms for multi-hop packet-radio networks. In *Proceedings of the 7th Annual Joint Conference of the IEEE Computer and Communcations Societies (INFOCOM) - Networks: Evolution or Revolution?*, pages 1110–1118, 1988.
26. Arumugam M, Kulkarni S. Self-stabilizing deterministic TDMA for sensor networks. In *Proceedings of the 2nd International Conference on Distributed Computing and Internet Technology (ICDCIT)*, pages 69–81. Springer, 2005.
27. Kulkarni S, Arumugam M. SS-TDMA: A self-

stabilizing MAC for sensor networks. In *Sensor Network Operations*, chapter 4, pages 186–218. IEEE Press, 2006.

28. Rhee I, Warrier A, Aia M, et al. Z-mac: a hybrid mac for wireless sensor networks. *IEEE/ACM Transactions on Networking (TON)*, 16(3):511–524, 2008.
29. Degesys J, Basu P, Redi J. Synchronization of strongly pulse-coupled oscillators with refractory periods and random medium access. In *Proceedings of the 2008 ACM symposium on Applied computing*, pages 1976–1980. ACM New York, NY, USA, 2008.
30. Degesys J, Nagpal R. Towards desynchronization of multi-hop topologies. In *Self-Adaptive and Self-Organizing Systems, 2008. SASO'08. Second IEEE International Conference on*, pages 129–138. IEEE, 2008.
31. Halkes G, Langendoen K. Crankshaft: An energy-efficient MAC-protocol for dense wireless sensor networks. *Wireless Sensor Networks*, 4373:228–244, 2007.
32. Zheng T, Radhakrishnan S, Sarangan V. Pmac: an adaptive energy-efficient mac protocol for wireless sensor networks. In *Parallel and Distributed Processing Symposium, 2005. Proceedings. 19th IEEE International*, pages 8–pp. IEEE, 2005.
33. Tjoa R, Chee K, Sivaprasad P, et al. Clock drift reduction for relative time slot TDMA-based sensor networks. In *15th IEEE International Symposium on Personal, Indoor and Mobile Radio Communications (PIMRC)*, volume 2, 2004.
34. Ganeriwal S, Kumar R, Srivastava M. Timing-sync protocol for sensor networks. In *Proceedings of the 1st International Conference on Embedded Networked Sensor Systems*, pages 138–149. ACM New York, NY, USA, 2003.
35. Maróti M, Kusy B, Simon G, et al. The flooding time synchronization protocol. In *Proceedings of the 2nd International Conference on Embedded Networked Sensor Systems*, pages 39–49. ACM, 2004.
36. Pussente R, Barbosa V. An algorithm for clock synchronization with the gradient property in sensor networks. *Journal of Parallel and Distributed Computing*, 69(3):261–265, 2009.
37. Elson J, Estrin D. Time synchronization for wireless sensor networks. In *Proceedings of the 15th International Parallel and Distributed Processing Symposium*, pages 1965–1970, 2001.
38. Liu M, Lai T, Liu M. Is clock synchronization essential for power management in IEEE 802.11-based mobile ad hoc networks? In *Proceedings from the Second IEEE International Conference on Mobile Ad Hoc and Sensor Systems*, 2005.
39. Mank S, Karnapke R, Nolte J. An adaptive TDMA based MAC protocol for mobile wireless sensor networks. In *Proceedings of the 2007 International Conference on Sensor Technologies and Applications*, pages 62–69. IEEE Computer Society, 2007.
40. Mank S, Karnapke R, Nolte J. MLMAC - An adaptive TDMA MAC protocol for mobile wireless sensor networks. In *Ad-Hoc & Sensor Wireless Networks: An International Journal, Special Issue on 1st International Conference on Sensor Technologies and Applications*, 2008.
41. Abramson N. The throughput of packet broadcasting channels. *IEEE Transactions on Communications*, 25(1):117–128, 1977.
42. Iwanicki K, van Steen M. Multi-hop cluster hierarchy maintenance in wireless sensor networks: A case for gossip-based protocols. In *Proceedings of the Sixth European Conference on Wireless Sensor Networks (EWSN 2009)*, pages 102–117, Cork, Ireland, 2009. Springer-Verlag LNCS 5432. URL `http://www.few.vu.nl/~iwanicki/publications/2009-02-EWSN/`.
43. Dobson M, Voulgaris S, van Steen M. Network-level synchronization in decentralized social ad-hoc networks. In *5th International Conference on Pervasive Computing and Applications (ICPCA)*, pages 206–212. IEEE, 2010.
44. Dobson M, Voulgaris S, van Steen M. Merging ultra-low duty cycle networks. *41st International Conference on Dependable Systems and Networks (DSN)*, pages 538–549, 2011.
45. Langendoen K, Baggio A, Visser O. Murphy Loves Potatoes: Experiences from a Pilot Sensor Network Deployment in Precision Agriculture. In *14th Int. Workshop on Parallel and Distributed Real-Time Systems (WPDRTS)*, 2006.
46. Pentland A. S. Automatic mapping and modeling of human networks. *Physica A: Statistical Mechanics and its Applications*, 378(1):59–67, 2007.
47. Varga A, Hornig R. An overview of the omnet++ simulation environment. In *Proceedings of the 1st international conference on Simulation tools and techniques for communications, networks and systems & workshops*, page 60. ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering), 2008.
48. Weingartner E, vom Lehn H, Wehrle K. A performance comparison of recent network simulators. In *IEEE International Conference on Communications (ICC)*, pages 1–5, 2009.
49. Köpke A, Swigulski M, Wessel K, et al. Simulating wireless and mobile networks in omnet++ the mixim vision. In *Proceedings of the 1st international conference on Simulation tools and techniques for communications, networks and systems & workshops*, page 71. ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering), 2008.
50. Ester M, Kriegel H, Sander J, et al. A density-based algorithm for discovering clusters in large spatial databases with noise. In *Pro-*

ceedings of the 2nd International Conference on Knowledge Discovery and Data Mining, pages 226–231. AAAI Press, 1996.

51. Waber B, Olguin Olguin D, Kim T, et al. Productivity through coffee breaks: Changing social networks by changing break structure. *Available at SSRN 1586375*, 2010.
52. Newman M. E. Spread of epidemic disease on networks. *Physical Review E*, 66(1):016128, 2002.
53. Eubank S, Guclu H, Kumar V. A, et al. Modelling disease outbreaks in realistic urban social networks. *Nature*, 429(6988):180–184, 2004.
54. Flajolet P, Martin G. N. Probabilistic counting algorithms for data base applications. *Journal of Computer and System Sciences*, 31(2):182–209, 1985.
55. Metwally A, Agrawal D, Abbadi A. E. Why go logarithmic if we can go linear?: Towards effective distinct counting of search traffic. In *Proceedings of the 11th international conference on Extending database technology: Advances in database technology*, EDBT '08, pages 618–629, New York, NY, USA, 2008. ACM. URL `http://doi.acm.org/10.1145/1353343.1353418`.
56. Datar M, Gionis A, Indyk P, et al. Maintaining stream statistics over sliding windows. *SIAM Journal on Computing*, 31(6):1794–1813, 2002.
57. Considine J, Li F, Kollios G, et al. Approximate aggregation techniques for sensor databases. In *Proceedings of the 20th International Conference on Data Engineering*, ICDE '04, pages 449–, Washington, DC, USA, 2004. IEEE Computer Society. URL `http://dl.acm.org/citation.cfm?id=977401.978068`.
58. Madden S, Franklin M, Hellerstein J, et al. Tag: A tiny aggregation service for ad-hoc sensor networks. *ACM SIGOPS Operating Systems Review*, 36(SI):131–146, 2002.
59. Baquero C, Almeida P, Menezes R, et al. Extrema propagation: Fast distributed estimation of sums and network sizes. *Parallel and Distributed Systems, IEEE Transactions on*, 23(4): 668–675, 2012.
60. Cichoń J, Lemiesz J, Zawada M. On cardinality estimation protocols for wireless sensor networks. *Ad-hoc, mobile, and wireless networks*, 6811:322–331, 2011.
61. Cichoń J, Lemiesz J, Szpankowski W, et al. Two-phase cardinality estimation protocols for sensor networks with provable precision. In *Wireless Communications and Networking Conference (WCNC), 2012 IEEE*, pages 2009–2013. IEEE, 2012.
62. Heidarian F, Schmaltz J, Vaandrager F. Analysis of a clock synchronization protocol for wireless sensor networks. *FM 2009: Formal Methods*, 5850:516–531, 2009.
63. Mirollo R, Strogatz S. Synchronization of pulse-coupled biological oscillators. *SIAM Journal on Applied Mathematics*, 50(6):1645–1662, 1990.

Patents

1. GMAC is protected by US Patent Application 12/215,040 and is available free of charge for academic use.

Summary

Recent advances in electronics have made wireless devices become smaller, lighter, less intrusive, and significantly cheaper: a commodity. This enables the deployment of increasingly larger collections of such devices for a multitude of applications, mainly for the collection of observed data (sensor networks). We expect wireless networks consisting of tens of thousands of nodes to be common in the near future. As these devices continue to decline in size and cost, we anticipate a proliferance of *wearable* wireless devices. Whether worn on or as an item of clothing, such a device can spontaneously form networks with similar devices worn by people in the vicinity of the device. These networks will therefore consist of many mobile devices, or *nodes*, and their topology will be strongly influenced by the social connections of the wearers. For this reason, we call them social ad hoc networks.

The potential uses of a social ad hoc networks are vast. Consider, for instance, a (large) group of people at a conference or similar social event, each wearing a small unobtrusive electronic badge with a limited radio range. By simply measuring how often and for how long two badges are within range of each other, we can register social interaction and study the structure of the social network. Furthermore, by aggregating and disseminating data we can even stimulate social interaction, for instance by a social game where groups of people (e.g., students of the same department) increase their score by talking to members of other groups, and lose points when sticking among themselves. Finally, a family or group of friends attending a large social event may be informed when they come in close proximity to each other, helping them to stay in contact. Other applications easily come to mind, including group-based messaging, finding people with specific profiles, and crowd management, to name a few. Furthermore, these devices need not be worn by people at all, but at the right price-point could be attached and used to identify physical objects. The topology of the resulting network could be used to establish groups of items that "belong" together, later notifying the user when the node attached to one of the items leaves the neighborhood of the others.

The primary focus of our investigation is on wireless communication protocols suitable for large social ad hoc networks. The domain of social ad hoc networks presents a number of challenges. First and foremost is the challenge of energy consumption. The wireless nodes that compose the network must be small and light enough to be comfortably worn, so large heavy batteries are out of the question. Nevertheless, the nodes must be capable of operating for prolonged periods without being recharged. In the setting of a large music festival, for example, the devices must operate continuously for several days. As the wireless radio is generally the most power-hungry component in such devices, the main onus for power savings is on the communication protocols. Duty cycling, or periodically turning a node's radio on and off, is the most effective method of conserving energy, but leads to the problem of synchronizing the active period's of the duty cycle. The second major challenge is node mobility. The fact that nodes exhibit arbitrary movement patterns means that

the network's topology is constantly changing. Due to this, nodes cannot use traditional routing algorithms, nor can they rely on symmetrical message exchanges. Finally, because a node cannot make any assumptions regarding its neighborhood, our solutions must be adaptable to changing conditions. For example, as a participant moves around the venue, her node may experience sudden changes in neighborhood size (and hence available bandwidth) depending upon the number of nearby participants.

In order to test potential solutions to the above challenges, we implement a simulator for our nodes using the OMNeT++ framework. Using our simulator, we can quickly evaluate new techniques without the time and expense of performing experiments with actual people and nodes. In this thesis, we explore a number of different synchronization techniques in both static and mobile networks. We simulate several different mobility patterns and a variety of network sizes, ranging from 64 nodes to over four thousand. In addition to analyzing network-level synchronization via simulations, we are also able to test out new applications. As an example, we evaluate a decentralized application designed to estimate the number of nodes participating in a network. An accurate method of estimating network size is a valuable tool for allowing our protocols to adapt to varying network conditions.

In addition to testing new protocol variations in simulation, we perform a number of real-world experiments. In these experiments, we examined the feasibility of using wearable sensor nodes as both a measurement tool and actuator for social behavior. To this end, we develop different applications to run on top of our networking protocols during actual social events. We deploy our social ad hoc networks in situations ranging from small, informal gatherings to large academic conferences with over two hundred attendees. From these experiments we can demonstrate the observation and measurement of social behavior in an unobtrusive manner. Through the use of gossip-based communication and a handful of passive nodes designed to observe and report the messages generated by the active badge nodes, we can visualize the dynamic topology of the entire social ad hoc network in real time. Furthermore, using a different application we are able to influence the behavior of the participants. This game is called *InCrowd*, and has very simple rules: a participant gains points for interacting with participants from a group other than their own. We typically group participants based on the department/company for which they work, or other similar demographic data. By measuring the inter- and intra-group interactions before the game is played, during the game, and after the game, we can recognize distinct changes in social behavior that persist after the game has ended.

The contributions of this thesis includes a thorough evaluation of communication techniques for social ad hoc networks and an investigation of the practicality of these techniques in real-world scenarios. The final solution presented in this work offers energy-efficiency, predictable battery usage regardless of network conditions (e.g., changes in mobility or density), and robust operation in the presence of failing nodes. We demonstrate the effectiveness of our solution in real-world networks consisting of over two hundred nodes, and provide simulations showing that they will scale to many thousands of nodes.

Samenvatting

Recente ontwikkelingen in elektronica hebben er voor gezorgd dat draadloze apparaten kleiner, lichter, minder indringend en aanzienlijk goedkoper zijn geworden. Dit maakt het uitzetten van een grotere verzameling van zulke apparaten als een netwerk van met elkaar communicerende knopen, mogelijk en inzetbaar voor verschillende doeleinden. Een typische toepassing is het verzamelen van observatie gegevens (in de vorm van zogeheten sensor netwerken). We verwachten dat draadloze netwerken bestaande uit duizenden knopen in de toekomst normaal zal zijn. Gezien de vermindering in omvang en kosten, anticiperen we op een groei van draagbare draadloze apparaten. Of het nu gaat om een object op een kledingstuk of dat deel uitmaakt van het kledingstuk, een dergelijk apparaat kan spontaan een netwerk vormen met vergelijkbare apparaten gedragen door mensen in zijn nabijheid. Deze netwerken zullen voornamelijk bestaan uit mobiele knopen, en hun topologie zal sterk beïinvloed worden door de sociale connecties van de dragers. Om deze reden noemen we ze sociale ad hoc netwerken.

Het potentieel van sociale ad hoc netwerken is groot. Stel je voor dat elke deelnemer aan een conferentie of vergelijkbaar evenement, een onopvallende elektronische badge met beperkt radiobereik draagt. Alleen al door het meten hoe vaak en hoe lang twee badges binnen elkaars bereik zijn, kunnen we sociale interactie registeren en de structuur van het sociale netwerk observeren. Bovendien kunnen we door het verzamelen en verspreiden van data sociale interacties stimuleren. Denk hierbij aan een sociaal spel waar groepen mensen (zoals studenten van dezelfde opleiding) hun score kunnen verbeteren door te praten met andere leden van andere groepen en punten verliezen als ze onder elkaar blijven. Een ander voorbeeld is dat van, een familie of groep vrienden die een groot sociaal evenement bezoeken. Zij kunnen geïnformeerd worden wanneer ze in elkaars buurt zijn, om zo in contact met elkaar te kunnen blijven, maar ook kunnen waarschuwingen verspreid worden zodra er een afdwaalt. Andere toepassingen zijn, groepsberichten, mensen vinden met specifieke profielen, en menigte beheer. Daarbij komt dat binnen afzienbare tijd, knopen niet expliciet als zodanig gedragen hoeven te worden door mensen, maar met een lage kostprijs ingebed kunnen worden bij andere objecten, zoals sieraden of mobiele telefoons.

De focus van het onderzoek is op de ontwikkeling van draadloze communicatieprotocollen die geschikt zijn voor grote sociale ad hoc netwerken. Het domein van sociale ad hoc netwerken brengt enige uitdagingen met zich mee. Een belangrijke uitdaging voor het onderhavige onderzoek is energiebeheer. Een draadloze knoop die deel uitmaakt van het netwerk moet klein en licht genoeg zijn om comfortabel te kunnen dragen. Het gebruik van batterijen van enige omvang is dan geen optie meer. Desalniettemin moet een knoop langdurig operationeel zijn zonder dat de batterij vervangen of opgeladen dient te worden. Zo zal voor een groot meerdaags muziekfestival, het sociale ad hoc netwerk ook meerdere dagen zonder verdere handmatige interventie moeten kunnen opereren.

Voor een knoop is de draadloze zender de grootste energieverbruiker. Energie kan dus al snel bespaard worden door een knoop een tijdlang niet te laten communiceren.

Periodiek de radio van een knoop uit te zetten, en telkens voor een relatief korte periode weer aan, is de meest effectieve manier om energie te besparen. Echter, dit aan-uit gedrag leidt tot synchronisatieproblemen: twee knopen kunnen tenslotte alleen maar communiceren als hun respectievelijke radios ook tegelijkertijd aan staan.

Een tweede grote uitdaging is de mobiliteit van knopen. Het feit dat een knoop ogenschijnlijk willekeurige bewegingspatronen heeft betekent dat de topologie van het netwerk constant verandert. In combinatie met de omvang van het netwerk, kunnen knopen niet traditionele oplossingen voor communicatie gebruiken, en in het bijzonder kan dikwijls niet symmetrisch berichten uitgewisseld worden. Daarbij komt dat niet alleen de topologie van het netwerk verandert, maar ook kan het aantal buren van een knoop sterk veranderen. Des te meer buren een knoop heeft, des te groter is de kans dat communicatie mislukt. Ook hier zal rekening mee gehouden moeten worden.

Om onze oplossingen te ontwikkelen en te toetsen aan deze uitdagingen, hebben we een simulator geïmplementeerd die gebruik maakt van het OMNeT++ raamwerk. Met onze simulator kunnen we snel nieuwe technieken evalueren zonder de tijd en kosten die het experimenteren met echte mensen en knopen met zich meebrengen. In dit proefschrift onderzoeken we een aantal verschillende synchronisatietechnieken in zowel statische als mobiele netwerken. We simuleren verschillende bewegingspatronen en verschillende netwerkgroottes variërend van 64 tot meer dan 4000 knopen. Naast het analyseren van onze oplossingen voor synchronisatie met behulp van simulaties hebben we ook nieuwe applicaties kunnen testen. Zo hebben we een applicatie gemaakt die inschat hoeveel knopen deelnemen in een netwerk. Een nauwkeurige methode om de grote van het netwerk in te schatten is een waardevol onderdeel om er voor te zorgen dat onze protocollen zich kunnen aanpassen aan variërende netwerkomstandigheden.

Naast simulaties, hebben we ook een aantal experimenten uitgevoerd om onze oplossingen te valideren. In deze experimenten hebben we de mogelijkheden onderzocht voor het gebruik van draagbare elektronische badges als zowel een meetinstrument als aandrijver voor sociaal gedrag. We hebben oplossingen geïmplementeerd voor een kleine informele bijeenkomsten tot grote academische conferenties met meer dan 200 bezoekers. Met deze experimenten konden we sociaal gedrag observeren en meten. Door het gebruik van zogeheten epidemische communicatie en een handvol louter waarnemende knopen die alleen maar het berichtenverkeer registreren, kunnen we de dynamische topologie van het gehele sociale ad hoc netwerk ter plekke visualiseren. Met een andere applicatie trachten we het gedrag van participanten te beïnvloeden. Dit spel, inCrowd geheten, gaat ervanuit dat elke deelnemer tot een specifieke groep behoort, zoals een afdeling, eenzelfde klas, enz. Het spel heeft een simpele regel: een participant krijgt punten voor interactie met andere participanten van een andere groep dan zijn eigen groep. Des te meer punten, des te meer er sprake is van sociale menging. Door de de interacties in en tussen groepen te meten, kunnen we aldus meten in hoeverre menging heeft plaatsgevonden.

De bijdragen van dit onderzoek bestaat uit de ontwikkeling en evaluatie van communicatietechnieken voor sociale ad hoc netwerken. Het omvat de uitvoerbaarheid

van deze technieken in levensechte scenarios. De uiteindelijk oplossingen gepresenteerd in dit werk zorgen voor energiezuinig en voorspelbare batterijverbruik ongeacht veranderende netwerkcondities, en een robuuste werking ook als knopen falen. We demonstreren de effectiviteit van onze oplossingen door levensechte experimenten met een paar honderd knopen, alsmede simulaties met netwerken van vele duizenden knopen.